Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels

UNIVERSITAT POLITÈCNICA DE CATALUNYA

TREBALL FINAL DE GRAU

TÍTOL DEL TFG: Ejemplos prácticos de aplicación de buses en aeronaves

TITULACIÓ: Grau en Enginyeria de Sistemes Aeroespacials

AUTOR: Daniel Cenador García

DIRECTOR: Josep M. Yúfera Gómez

DATA: 7 de febrer de 2020

Título: Ejemplos prácticos de aplicación de buses en aeronaves

Autor: Daniel Cenador García

Director: Josep M. Yúfera Gómez

Fecha: 7 de febrero de 2020

Resumen

El objetivo de este trabajo es estudiar la manera de crear una representación de una red aviónica, para poder modificarla y capturar resultados con el fin de analizar los comportamientos de la red, y crear ejemplos y ejercicios para su uso en el estudio de los protocolos de comunicación aeronáuticos. Los protocolos de comunicación escogidos para este proyecto son TTEthernet (Time-Triggered Ethernet) y AFDX (Avionics Full Duplex Switched Ethernet), dos protocolos basados en el estándar Ethernet IEEE 802.3, full duplex, deterministas, con topología de estrella y con altas tasas de transmisión que ayudan a reducir el peso del cableado comparado con otros estándares utilizados en la aviónica.

Para la creación de la aplicación se ha utilizado el software informático OMNeT++, un simulador modular y extensible de redes basado en el lenguaje de programación C++, con representación gráfica de la red simulada a tiempo real, que permite pausar la simulación para comprobar el estado de los nodos, o el contenido de los mensajes.

El trabajo se divide en dos partes, una primera que trata el primer protocolo TTEthernet, y la segunda parte trata AFDX. En cada parte se explica el funcionamiento del estándar pertinente, cuáles son sus características, y que estrategias usa para adaptar Ethernet al mundo aeronáutico. A continuación se explica su implementación en OMNeT++, las pruebas realizadas sobre esa red creada, análisis de los resultados recogidos por el sistema y posibles ejercicios relacionados la simulación. Seguido de estos dos capítulos hay unas conclusiones generales sobre el proyecto seguidas por un anexo. En el anexo se detalla información que ayuda a poner en contexto el trabajo, series de capturas de las simulaciones y código utilizado en para crear la simulación de la red aeronáutica de comunicaciones.

Title: Practical examples of buses applications in aircrafts

Author: Daniel Cenador García

Director: Josep M. Yúfera Gómez

Date: February 7th, 2020

Overview

The objective of this project is to study how to build an avionics communication system application, being able to modify it and capture the results in order to analyse the behaviour of the network, with the intention of create examples and tests for the academic purpose of studying aeronautic communication protocols. The protocols studied in this project are TTEthernet (Time-Triggered Ethernet) and AFDX (Avionics Full Duplex Switched Ethernet), both of them based in IEEE 802.3 Ethernet standard, being full duplex, deterministic, using star-topology, with high transmission rates and less weight with the use of less cable compared with older avionics communication standards.

The application is build using OMNeT++ informatic software, an extensible and modular network simulator based on C++ programming language. This simulator features a graphic representation of the network simulated on real time, being able to pause the simulation and check messages content and devices status.

The project is divided into two different parts, the first one contains the work related with TTEthernet and the second one with AFDX. Each part explains the basics of the protocol, like characteristics, modifications to Ethernet standard to make it compatible with aeronautics requirements, word structure, etc. Next is explained how is created a network using this protocol in OMNeT++, the test performed, the analysis of the gathered data and an approach to create practical exercises with the simulation. Following these two parts there are a general conclusion of the project. In the annex there is information useful to understand the background of the project, a series of captures of the simulation and code used to create the network studied.

ÍNDICE

ÍNDICE DE FIGURAS

INTRODUCCIÓN

Para entender la necesidad de estudio de los protocolos de comunicación AFDX y TTEthernet en el mundo aeronáutico, es necesario entender el contexto[1]. A finales del siglo XX, los aviones seguían utilizando muchas señales analógicas, y de las pocas que eran digitales utilizaban estándares de comunicación como ARINC 429, el cual tiene una topología de bus, es simplex y con bajas tasas de transmisión (de 12,5 a 100 kbps). Esta comunicación limita el formato del dato, impide la aplicación de protocolos complejos y aporta un gran peso al avión al necesitar varios kilómetros de cable para hacer todas las interconexiones entre los sistemas aviónicos, del orden de 200 km de cable en aviones *wide-body.*

Mover información entre sistemas aviónicos se ha convertido en una necesidad crucial, y los sistemas de transmisión de datos electrónicos son la solución. Desde la entrada del avión Airbus 320 en 1988, los aviones controlados electrónicamente o *fly-by-wire,* se ha convertido en el sistema de control preferido para los nuevos aviones comerciales, por ejemplo, la línea de Airbus 350. *Fly-by-wire,* es como se denomina a aquellos aviones los cuales han sustituido los controles de vuelo manuales, sistemas manuales o hidráulicos, con una interfaz electrónica. La principal ventaja de este sistema es la ligereza, además de poseer una mayor fiabilidad y tolerancia al daño. También proporciona mayor control de la aeronave al utilizar una computadora para asegurar la estabilidad de la nave durante las operaciones.

También hay más subsistemas electrónicos en los aviones comerciales, como plataformas inerciales para la navegación, sistemas de control, sensores, sistemas de comunicación, la caja negra (DFDR) … Todos ellos tienen que ser provistos de una gran fiabilidad, de tasas de transmisión altas y comunicación en tiempo real.

Aquí es donde entran AFDX y TTEthernet, dos estándares de comunicación basados en Ethernet, con las capacidades necesarias para el uso en el mundo aeronáutico.

El objetivo de este trabajo es construir una red utilizando los protocolos de comunicación aeronáuticos anteriormente nombrados, AFDX y TTEthernet, y modificar su configuración y recoger datos para obtener unos resultados sobre las modificaciones. Estos ejemplos se enfocan desde un punto de vista académico, donde se busca analizar los efectos en las diferentes configuraciones de redes AFDX y TTEthernet, en como configurar los nodos y sus efectos. La elección de estos protocolos se efectuó por ser los más novedosos dentro del sector. Otras opciones eran ARINC 429, un protocolo muy simple y robusto, con el cual era muy difícil modificar sus comportamientos dada su simpleza y poca configurabilidad, o el bus CAN, un bus muy utilizado en otras industrias, como la automovilística, descartado por ya tener suficientes estudios. Además, AFDX es un protocolo propio de la industria aeronáutica, y TTEthernet es utilizado en la industria aeroespacial para la confección de satélites.

El documento se estructura en dos capítulos, cada uno reservado para un protocolo, seguido de las conclusiones. Dentro de cada capítulo se trata el funcionamiento de cada protocolo, con que herramientas se crearán las recreaciones de estos protocolos, el análisis de los resultados obtenidos y finalmente el planteamiento de posibles ejercicios creados a partir de la simulación. El apartado práctico de este proyecto ha sido realizado íntegramente con software de simulación de redes. Al inicio de proyecto se planteó la recreación de una red aviónica con hardware propio del protocolo, pero el precio de los componentes era demasiado caro para el presupuesto del proyecto (del orden de miles de euros los equipos terminales y switches para laboratorio).

La motivación de este proyecto se debe a la relación entre el mundo aeronáutico y el de las telecomunicaciones, dos de las carreras que se cursan en nuestra facultad, y a la posibilidad de que la aplicación de estos protocolos pueda ser aprovechada para la formación aviónica de futuros alumnos.

CAPÍTULO 1. TTEthernet

1.1 Time Triggered Ethernet

Time Triggered Ethernet, denominado TTEthernet o TTE, es un estándar de comunicación desarrollado por *RealTime System Group* en la Universidad de Viena el año 2000. TTE combina el estándar basado en comunicación *Best Effort Ethernet* IEEE 802.3, una comunicación basada en eventos sin limitación de ancho de banda, modificado para añadir tráfico *Time Triggered,* manteniendo la estructura de la red. El tráfico *Time triggered* está basado en tiempo, en vez de eventos, por lo que cada mensaje tiene un intervalo de tiempo predefinido para ser enviado. TTE añade al modelo encabezados *Time triggered* que necesita de una sincronización global de todos los sistemas de la red para transmitir los mensajes en el tiempo exacto.

TTEthernet aumenta el QoS o calidad de servicio (*Quality of Service*, en inglés) al utilizar tres tipos de mensaje dentro de Ethernet. El primer tipo son los mensajes TT (*Time-Triggered*), que toman prioridad sobre los demás mensajes y son enviados durante su ventana temporal previamente definida, pensados para la transmisión en aplicaciones críticas presentes en aviones *fly-by-wire* que, mediante el uso de impulsos eléctricos, transmiten las instrucciones que el piloto realiza en los mandos de cabina, a las superficies de control aerodinámicas que se encargan del movimiento del avión, por ejemplo, un cambio de posición de *flaps.* El segundo tipo son los mensajes RC (*Rate-constrained*), que tiene un ancho de banda asegurado, pero no está sincronizado, por lo que sigue estando basado en eventos. Este tipo de comunicación puede ser aceptable para aplicaciones menos críticas en el tiempo de recepción como la retracción del tren de aterrizaje. Finalmente tenemos los mensajes BE (*Best-Effort*) que utilizan el ancho de banda restante: es el estándar Ethernet que no provee de garantías en latencia. En la fig. 1.1 se observa cómo se ordenan los tipos mensajes en el switch respetando sus respectivas prioridades (TT>RC>BE).

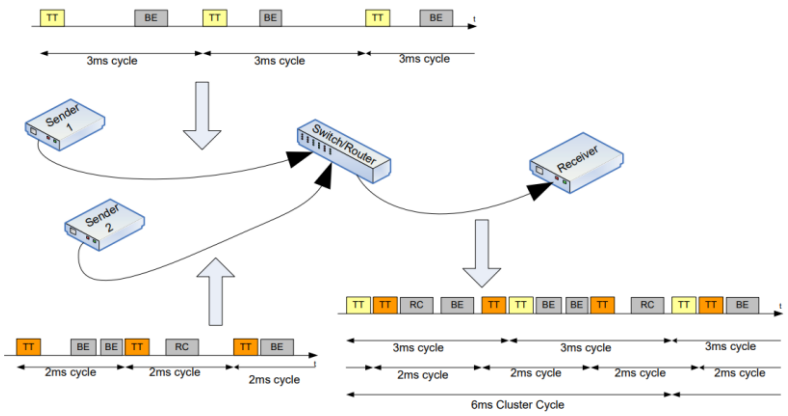


Fig. 1.1 Comportamiento de los tres tipo de tráfico

TTE está publicado como SAE AS6802 y la estandarización IEEE está en progreso. Es compatible con las capas físicas de Ethernet, soporta sus diferentes anchos de banda (10/100/1000 Mbps) e integra con todos los componentes de una red Ethernet. TTE también admite el montaje con redundancia, en el cual los nodos están conectados a dos redes gemelas, donde transmitirán el mismo mensaje de manera sincronizada, para alcanzar la probabilidad de error de 10^{-9} necesaria en los sistemas críticos aeronáuticos (ver fig. 1.2).

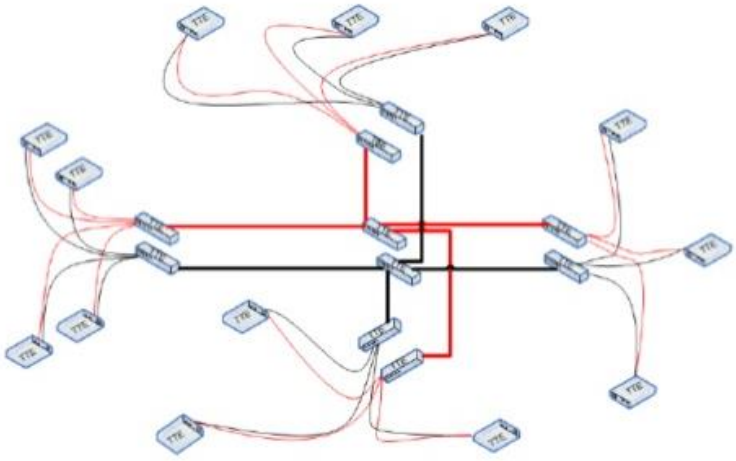


Fig. 1.2 Redundancia en la red TTEthernet

El trafico TT y el RC utilizan *Virtual Links* (VL) que reemplazan el uso de la MAC para identificar el destino. Los enlaces VL simulan conexiones simplex, con un solo emisor y varios receptores, dentro de una red más compleja, un concepto parecido al protocolo ARINC 429 (ver anexo 5.1 para la definición de ARINC 429), con la diferencia de que no existen de manera física, sino lógica. Los mensajes llevarán en su cabecera un campo de *Virtual Link Identifier* (VLID) y este valor se utilizará para redirigir el mensaje dentro de la red a los nodos destinatarios. A su vez, si un nodo recibe un mensaje con un VLID no correspondiente, lo eliminará. Otra aplicación de los VL es predefinir el camino que seguirá un mensaje entre el origen y el destino, para eliminar ambigüedades.

Los VL limitan la tasa de transmisión de cada canal, para evitar el colapso del canal, mediante un parámetro llamado *Bandwith Allocation Gap* o BAG. Este BAG es el tiempo mínimo que tiene que pasar para que el emisor pueda volver a enviar un mensaje (ver figura 1.3) y puede tener los siguientes valores: 1, 2, 4, 8, 16, 32, 64 y 128 milisegundos. Normalmente, el BAG se asigna de manera proporcional a la carga de los mensajes, para repartir el ancho de banda del canal de manera equitativa entre VL.

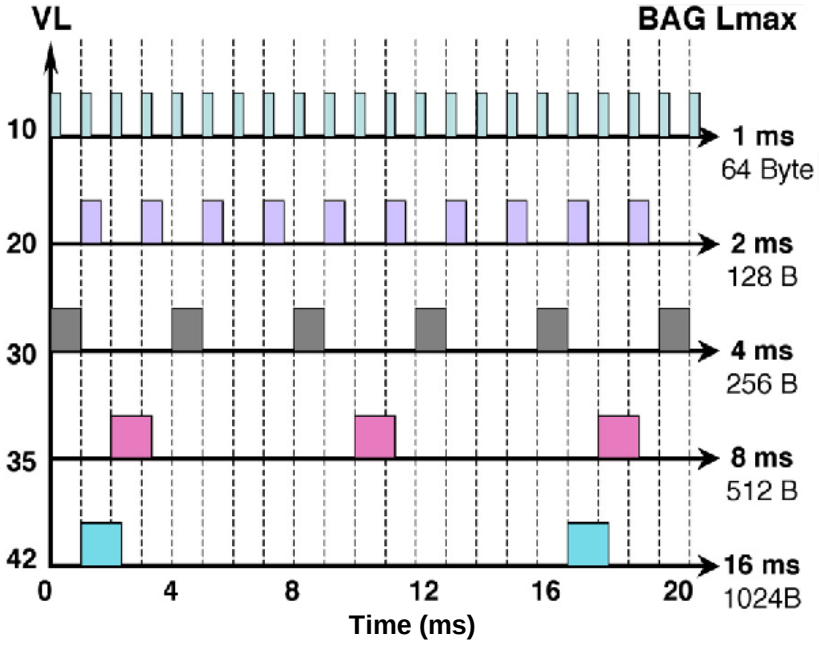


Fig. 1.3 Aplicación del parámetro BAG

1.1.1 Tres tipos de tráfico: TT + RC + BE

TTEthernet es compatible con tres tipos de tráfico: tráfico *time-triggered* (TT), tráfico *rate-contrained* (RC) y *best-effort* (BE). El comportamiento del tráfico RC es idéntico al protocolo AFDX, haciendo TTE compatible con AFDX.

1.1.1.1 Comunicación *Time-Triggered*

TT contiene el tráfico que ha sido transmitido acorde a un paradigma de comunicación *time-triggered,* en el cual cada mensaje se envía en su *time slot* o ventana temporal. Las comunicaciones TT requieren de una sincronización entre los sistemas emisores. Para establecerla y mantenerla se utilizan tramas de control que se van enviando periódicamente los nodos de la red. Estas tramas de control son compatibles con el estándar Ethernet. El tráfico TT utiliza los enlaces VL para el encaminamiento de mensajes.

Cada nodo tiene guardado una tabla en la cual están estipulados los tiempos en los cuales pueden transmitir el mensaje. Esta tabla se configura *offline* y tiene que garantizar que no hay colisión entre los mensajes TT. El problema de definir esta tabla es que se complica drásticamente cuando la red tiene muchos nodos o cuando la latencia buscada es muy pequeña.

Una de las ventajas del tráfico TT es la preplanificación del intercambio de mensajes, al estar estipulado cuando va a transmitir cada sistema, la latencia y el *jitter* son reducidos drásticamente ya que ningún dispositivo se pelea por el medio.

Por el otro lado, las comunicaciones TT requieren una sincronización de reloj y de un algoritmo de inicio. Esto conlleva al envío de mensajes de sincronización por parte del nodo responsable de la sincronización provocando una pérdida en el ancho de banda de la red. Este tipo de comunicación también dificulta la adición de nuevos nodos, ya que conllevaría reestructurar la tabla con nuevos tiempos de mensaje. Por eso muchas tablas están ya creadas con *slots* vacíos para ser utilizados por nuevos nodos en el futuro, y mientras no son utilizados son aprovechados por los otros dos tipos de tráfico.

1.1.1.2 Comunicación *Rate-Constrained*

RC se refiere a todo el tráfico transmitido dentro del paradigma de comunicación *Rate-Constrained.* Este tráfico está basado en eventos y utiliza VLs. A diferencia del TT, el tráfico RC no está reñido por una tabla preconstruida: cada transmisor adapta el tráfico a cada flujo, asegurándose de que habrá un mínimo de espacio entre dos mensajes enviadas en el cual se meterá un paquete RC. Los switch monitorizan la red y eliminarán cualquier mensaje RC que haya sido enviado demasiado rápido, no respetando el espacio mínimo entre tramas. En la configuración del tráfico RC hay que definir un valor de prioridad por VL, para el caso de que dos mensajes RC lleguen al mismo tiempo al switch, poder calcular cual enviar primero.

La tasa de transmisión en RC está limitada por el BAG. Como los nodos no están sincronizados, los picos de carga tienen que ser considerados para calcular el tamaño de los buffers de los nodos para evitar pérdida de datos. Como la tasa y el tamaño de los buffers son prioritarios, la latencia y el jitter pasan a segundo plano en el diseño.

En la escalabilidad, añadir nodos al tráfico RC es más fácil en comparación con el tráfico TT, aun así, requeriría un recálculo de la memoria de los buffers y de la latencia para garantizar que los mensajes siguen llegando. También se puede aplicar el mismo anticipo que con el tráfico TT, reservando ancho de banda para futuras extensiones de la red.

1.1.1.3 Comunicación *Best-Effort*

BE es un paradigma de la comunicación en el cual la red trata por igual todos los paquetes de datos, sin dar prioridad a ninguno. Esto provoca que no haya garantías, y en una sobrecarga de la red, el switch tiraría los paquetes BE que superen el ancho de banda disponible.

Los mensajes BE pueden ser utilizados para servicios opcionales o no críticos, entre los cuales se encontrarían las aplicaciones relacionadas con la cabina de pasajeros, por ejemplo, el control de temperatura, megafonía, indicador de cinturón, monitores, etc.

Añadir nodos al tráfico BE no supone ninguna reconfiguración, pero limita el ancho de banda disponible, augmentando la probabilidad de picos de sobrecarga en la red.

1.2 Simulador OMNeT++

Para hacer pruebas con una red TTEthernet contaremos con OMNeT++, un simulador modular y extensible de redes basado en el lenguaje de programación C++, que proporciona el esqueleto para crear simulaciones discretas de redes. OMNeT++ tiene una arquitectura genérica que permite simular redes inalámbricas o por cable, analizar protocolos, y modelar cualquier sistema basado en el intercambio de mensajes. Esto lo consigue utilizando el *framework INET,* una librería de acceso libre que provee de protocolos (TCP, UDP, IPv4, IPv6, OSPF, BGP, etc.) y modelos (Ethernet, PPP, IEEE 802.11, etc), para investigadores y estudiantes trabajando con redes de comunicación.

Ofrece un IDE (*Integrated Development Enviroment*) de Eclipse, con el que representa gráficamente las redes simuladas, permitiendo pausar la simulación para observar el contenido de los mensajes transmitidos en ese mismo instante, el estado de los módulos de la red, etc. Además, admite la instalación de paquetes de terceros para simular redes y protocolos no incluidos en el paquete básico de INET. La página oficial de OMNeT++[2] tiene recogida una selección de paquetes de terceros en los que se pueden encontrar protocolos utilizados en el mundo aeronáutico, tales como AFDX o TTEthernet.

Los requisitos para este simulador son mínimos: al tener un núcleo de simulación C++ estándar, se puede ejecutar en cualquier plataforma que tenga un compilador C++, Linux, Windows o MacOS. El simulador es de libre acceso para uso académico o no lucrativo.

Las principales razones por las que se ha escogido el simulador OMNeT++ de entre otros (tales como *GNS3, Cisco Paquet Tracer* o *Boson Netsim*) para este proyecto son su compatibilidad con la mayoría de los sistemas operativos, ser de libre de acceso para el uso académico y que utiliza el lenguaje C++, el cual es temario obligatorio del grado.

Para instalar OMNeT++ hay que descargar el programa de su página web[3] y la descarga varía dependiendo del sistema operativo. Su instalación no es muy intuitiva, pero hay una guía de instalación[4] que indica paso por paso el proceso.

1.3 Paquete CoRE4INET

CoRE4INET es una extensión del *framework* de INET que recoge los protocolos para la simulación de Ethernet en tiempo real incorporando los protocolos AS6802 (TTEthernet) y *Audio Video Bridging* (AVB), de entre otros. Esta extensión ha sido creada por el grupo de investigación CoRE (*Communication over Real-time Ethernet*)[5] que trabaja en el estudio de soluciones para la comunicación crítica basada en tiempo utilizando la tecnología Ethernet.

1.3.1 Instalación

Su instalación se puede hacer a través del IDE de OMNeT++. Para ello hay que ir a *Help, Install New Software…* y en la ventana que aparece hay que ir a *Add,* introducir la dirección *http://sim.core-rg.de/updates/* e instalar los paquetes *Abstract Network Description Language, CoRE Simulation Model Installer* y *Gantt Chart Timing Analyzer.*

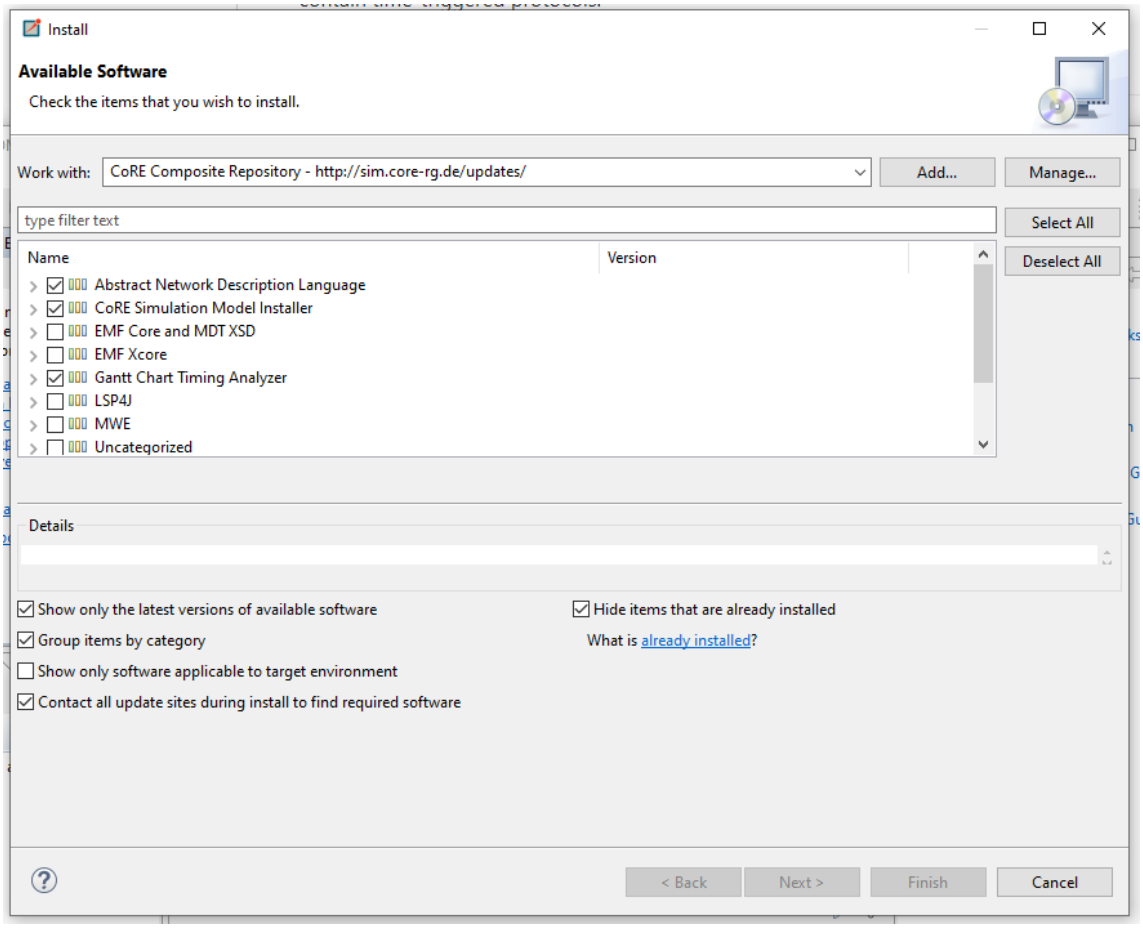


Fig. 1.4 Ventana de instalación de software

Una vez se hayan descargado estos tres paquetes hay que entrar otra vez en *Help* e *Install CoRE Simulation Models…* y instalar el simulador CoRE4INET. Para que funcione tambien se necesita el paquete INET basico de OMNeT++, pero la version 3.6 (actualmenta va por la 4.1.2). Esta se puede descargar de la pagina oficial de INET[6]. Una vez descargado y descomprimido, hay que depositarlo en el fichero con el que trabaja OMNeT++ (la primera vez que se abre el IDE de OMNeT++ te pide seleccionar un fichero para generar un entorno de trabajo). Para instalarlo se tiene que entrar en el IDE, ir a *File, Import…;* seleccionar *General, Existing Projects into Workspace*, y seleccionar *inet.*

1.3.2 Creación de una red con CoRE4INET

Para crear una red TTEthernet primero hay que crear una carpeta dentro de la carpeta de CoRE4INET que se utilizara como entorno de trabajo. Para que el IDE reconozca esta carpeta y deje configurar gráficamente la red necesita ser un *NED source folder,* directorios utilizados por el simulador para identificar donde hay una red. Para configurar los *NED source folder* hacer clic derecho en la carpeta del paquete CoRE4INET, selecciona propiedades, y dentro de esta ventana abrir la pestaña OMNeT++, seleccionar *NED source folder* y marcar la carpeta previamente creada.

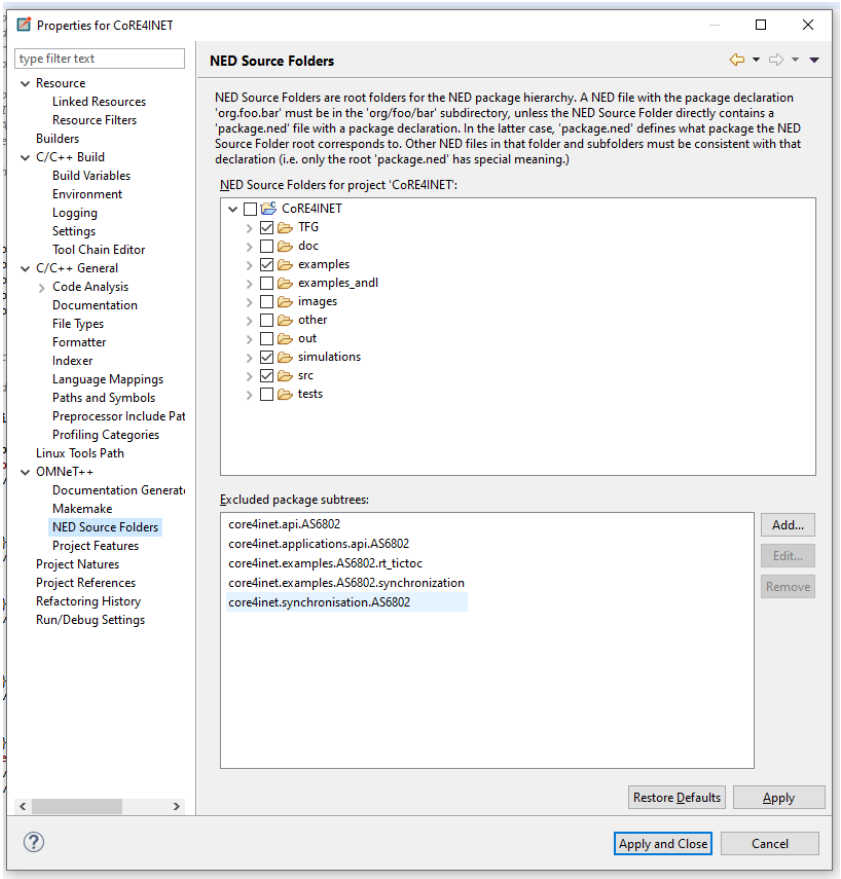


Fig. 1.5 Configuración de los *NED Source Folders*

Una vez creado el entorno de trabajo podemos empezar a montar la red. Primero creamos dentro de nuestro entorno de trabajo un archivo NED. Los archivos NED (*NEtwork Description*) son un tipo de archivo que utiliza OMNeT++ para establecer la red, y permite su configuración mediante código o gráficamente. Nos aparecerá una hoja en blanco con un texto tabulado con términos de licencias de OMNeT.

El primer paso es importar los módulos que vamos a utilizar en este archivo, que en este caso es la conexión ethernet a 100 Mbps de INET. Los módulos son clases C++ que definen un componente físico de la red, como, por ejemplo, una conexión ethernet.

Ahora podemos definir los nodos y las conexiones entre ellos. Crearemos una red (*RED*) sencilla con dos nodos (*nodo1 y nodo2*) y un switch (*switch1*), para minimizar las complicaciones de la creación. El nodo 1 enviará mensajes TT a través del virtual link 100 al nodo 2, y recibirá mensajes RC del nodo 2 por el virtual link 101. Ambos nodos enviarán y recibirán mensajes BE.

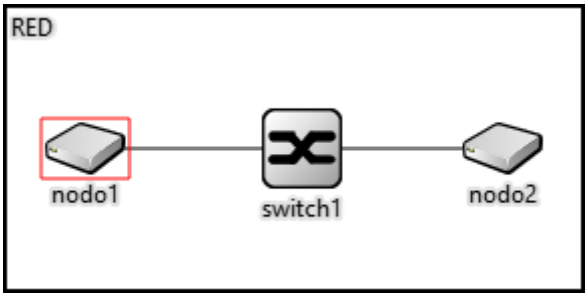


Fig. 1.6 Esquema de la red

1.3.2.1 Definición de la red

Definimos RED como una clase C++, y dentro de ella especificamos que tendrá dos nodos y un switch. Hay que especificar los puertos que tendrá el switch y cómo estarán hechas las conexiones, incluidas longitudes. Los parámetros que empiezan por *@display* son gráficos y OMNeT los configura automáticamente si se cambian desde el apartado de edición gráfica de la red. Los errores que aparecen se deben a que aún no hemos definido los nodos.

```
//módulos a importar
import inet.node.ethernet.Eth100M;
//definición de la red
network RED
{
    @display("bgb=,,white");
    submodules: //nodos y switches que compondran la red
        nodo1: nodo1 {
            @display("p=39,70");
        }
```

```
        nodo2: nodo2 {
            @display("p=247,70");
        }
        switch1: Switch1 {
            parameters:
                @display("p=147,70");
            gates:
                ethg[2];//vector con los puertos del switch
        }
    connections:
       //Conexiones de dispositivos dentro de la red
        nodo1.ethg <--> Eth100M {  length = 10m; } <--> switch1.ethg[0];
        nodo2.ethg <--> Eth100M {  length = 10m; } <--> switch1.ethg[1];
}
```

Código 1.1 Estructura de la red

Para definir el nodo 1 tendremos que crear un nuevo archivo NED e importar los módulos de CoRE4INET, diferentes a los anteriores. Los nodos tienen una parte común entre sí, pero hay que añadir submódulos específicos para cada VL, hay que definir un buffer y un controlador, que valida la integridad del paquete, por cada VL. Estos submódulos son iguales para entrada y salida de mensajes, además, estos submódulos son específicos de cada tráfico y hay que conectarlos entre ellos.

```
//módulos a importar
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;
//definición del nodo
module nodo1 extends TTEEtherHost //TTEEtherHost es el módulo predefinido de
CoRE4INET para un nodo TTE
{
    @display("bgb=503,314");
    submodules: //dispositivos internos del nodo
        //controlador del trafico TT
        vl_100_ctc: TTIncoming {
            parameters:
                @display("p=350,60");
        }
        //Buffer del VL
        vl_100: TTDoubleBuffer {
            parameters:
                @display("p=450,60");
        }
       //controlador del trafico RC
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=350,160");
        }
       //Buffer del VL
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=450,160");
```

```
        }


    connections: //conexiones entre el controlador y el buffer
        vl_100_ctc.out --> vl_100.in;
        vl_101_ctc.out --> vl_101.in;
}
```

Código 1.2 Estructura del nodo 1

```
//módulos a importar
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;
//definición del nodo
module nodo2 extends TTEEtherHost //TTEEtherHost es el módulo predefinido de
CoRE4INET para un nodo TTE
{
    @display("bgb=512,314");
    submodules: //dispositivos internos del nodo
       //controlador del tráfico TT
        vl_100_ctc: TTIncoming {
            parameters:
                @display("p=350,60");
        }
        //Buffer del VL
        vl_100: TTDoubleBuffer {
            parameters:
                @display("p=450,60");
        }
       //controlador del tráfico RC
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=350,160");

        }
       //Buffer del VL
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=450,160");
        }
    connections: //conexiones entre el controlador y el buffer
        vl_100_ctc.out --> vl_100.in;
        vl_101_ctc.out --> vl_101.in;
}
```

Código 1.3 Estructura del nodo 2

El archivo NED del switch tiene la misma estructura que los de los nodos y también necesita una entrada y un buffer por cada VL con la que trabaja.

```
//módulos a importar
import core4inet.nodes.ethernet.AS6802.TTEEtherSwitch;
```

```
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;

//definición del switch
module Switch1 extends TTEEtherSwitch //TTEEtherSwitch es el módulo
predefinido de CoRE4INET para un switch TTE
{
    @display("bgb=500,312");
    submodules: //dispositivos internos del switch
      //controlador del tráfico TT
        vl_100_ctc: TTIncoming {
            parameters:
                @display("p=349,60");
        }
      //Buffer del VL
        vl_100: TTDoubleBuffer {
            parameters:
                @display("p=439,61");
        }
      //controlador del tráfico RC
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=349,140");
                hardware_delay = hardware_delay;
        }
      //Buffer del VL
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=439,141");
        }
    connections: //conexiones entre el controlador y el buffer
        vl_100_ctc.out --> vl_100.in;
        vl_101_ctc.out --> vl_101.in;
}
```

Código 1.4 Estructura del switch

En resumen, el archivo NED de la red define que nodos y switches va a tener y cómo van a estar conectados. Los archivos NED de los nodos definirán la estructura de estos, necesitado submódulos específicos para cada VL que vaya a trabajar y son iguales para emisión y recepción, para el tráfico BE no se necesita un submódulo extra. Los NED de switches son parecidos a los de nodos, cambiando el módulo que llama, y también necesita de los submódulos para trabajar con cada VL. Con estos cuatro archivos NED tenemos la estructura básica de la red, pero faltarían los archivos INI para configurarla.

1.3.2.2 Configuración de la red

Primero crearemos un archivo INI (*Initialization File*) en nuestro entorno de trabajo. En estos archivos se especifican los valores concretos que queremos

utilizar como parámetros de entrada, la generación de números aleatorios o el tiempo límite de simulación y se necesita un archivo INI por cada NED.

Para el nodo 1 se definirá la MAC 0A-00-00-00-00-00 y cuatro aplicaciones. La aplicación 0 generara mensajes TT por el VL 100. Dentro de esta aplicación se define cada cuanto la aplicación genera un mensaje, la ventana de transmisión del mensaje TT o *time slot* para el controlador, si el mensaje de la aplicación entra en el controlador fuera de esa venta se elimina. También se especifica la carga del paquete, el VLID, el destino (la tarjeta física para los mensajes generados) y la frecuencia de transmisión (cada cuanto se envía un mensaje). Los tiempos definidos están definidos a partir del inicio de ciclo que se repite indefinidamente, la duración del ciclo se especifica en el código 1.8. La aplicación 1 se encarga de recibir los mensajes RC, se le especifica el controlador que verifica que los paquetes recibidos son correctos, buffer, VLID y el destino (la aplicación para los mensajes recibidos). La aplicación 2 recibe los mensajes BE y se configura el destino. Finalmente, la aplicación 3 se encarga de generar tráfico BE. Hay que indicar cuanta carga llevan esos mensajes y cada cuanto se generan.

```
[General]
network = RED
//dirección MAC
**.nodo1.phy[0].mac.address = "0A-00-00-00-00-00"

**.nodo1.numApps = 4
**.nodo1.app[0].typename = "TTTrafficSourceApp"//aplicación de transmisión TT
**.nodo1.app[0].displayName = "vl_100"
**.nodo1.app[0].action_time = 980us
**.nodo1.app[0].payload = 46Byte
**.nodo1.app[0].ct_id = 100// VLID de la aplicación
**.nodo1.app[0].buffers = "vl_100"
//Inicio time slot del controlador
**.nodo1.vl_100_ctc.receive_window_start = sec_to_tick(970us)
// final time slot del controlador
**.nodo1.vl_100_ctc.receive_window_end = sec_to_tick(990us)
**.nodo1.vl_100_ctc.permanence_pit = sec_to_tick(990us)
**.nodo1.vl_100.destination_gates = "phy[0].TTin"
**.nodo1.vl_100.ct_id = 100//VLID a controlar
// cuando envia el mensaje la aplicacion
**.nodo1.vl_100.sendWindowStart = sec_to_tick(1000us)
**.nodo1.phy[0].shaper.tt_buffers = "vl_100"


**.nodo1.app[1].typename = "CTTrafficSinkApp""//aplicación de recepción RC
**.nodo1.app[1].displayName = "vl_101"
**.nodo1.phy[0].inControl.ct_incomings = "vl_101_ctc"//VLID de la aplicacion
**.nodo1.vl_101_ctc.bag = sec_to_tick(880us) //BAG
**.nodo1.vl_101.ct_id = 101//VLID del controlador
**.nodo1.vl_101.destination_gates = "app[1].RCin"


**.nodo1.app[2].typename = "BGTrafficSinkApp"//app de recepción BE
**.nodo1.bgIn.destination_gates = "app[2].in"
```

```
**.nodo1.app[3].typename = "BGTrafficSourceApp”//app de emisión BE
**.nodo1.app[3].payload = intWithUnit(uniform(1500Byte, 1500Byte))
**.nodo1.app[3].sendInterval = uniform(200us,500us)
**.nodo1.app[3].destination_gates = "phy[0]"
```

Código 1.5 Archivo INI del nodo 1

Para el nodo 2 se definirá la MAC 0A-00-00-00-00-00 y cuatro aplicaciones. La aplicación 0 generara mensajes BE por el VL 101, dentro de esta aplicación se configura la carga de los mensajes, el BAG, el VLID, buffers, el destino y la prioridad. La aplicación 1 se encarga de recibir los mensajes TT. Se le especifica el controlador de los mensajes TT, la ventana de tiempo de recepción de los mensajes TT, el VLID y el destino. Las aplicaciones 2 y 3, que se encargan del tráfico BE, son iguales que en el nodo 1.

Para los tiempos de recepción del mensaje TT hay que tener en cuenta el tiempo de propagación de los mensajes. En el paquete CoRE4INET no especifica de manera clara cuál es la velocidad de propagación del mensaje por el cable, pero al tener ejemplos ya construidos, he podido obtener que para conexiones de 10m, el tiempo de transmisión es de 25 microsegundos.

```
[General]
network = RED

**.nodo2.phy[0].mac.address = "0A-00-00-00-00-01"

**.nodo2.numApps = 4
**.nodo2.app[0].typename = "RCTrafficSourceApp"
**.nodo2.app[0].displayName = "vl_101"
**.nodo2.app[0].interval = 1ms
**.nodo2.app[0].payload = 46Byte
**.nodo2.app[0].ct_id = 101
**.nodo2.app[0].buffers = "vl_101"
**.nodo2.vl_101.destination_gates = "phy[0].RCin"
**.nodo2.vl_101.bag = sec_to_tick(900us)
**.nodo2.vl_101.priority = 0
**.nodo2.vl_101.ct_id = 101

**.nodo2.app[1].typename = "CTTrafficSinkApp"
**.nodo2.app[1].displayName = "vl_100"
**.nodo2.vl_100_ctc.receive_window_start = sec_to_tick(1020us)
**.nodo2.vl_100_ctc.receive_window_end = sec_to_tick(1040us)
**.nodo2.vl_100_ctc.permanence_pit = sec_to_tick(1040us)
**.nodo2.vl_100.ct_id = 100
**.nodo2.vl_100.sendWindowStart = sec_to_tick(1080us)
**.nodo2.vl_100.destination_gates = "app[1].TTin"
**.nodo2.phy[0].inControl.ct_incomings = "vl_100_ctc"

**.nodo2.app[2].typename = "BGTrafficSinkApp"
**.nodo2.bgIn.destination_gates = "app[2].in"

**.nodo2.app[3].typename = "BGTrafficSourceApp"
**.nodo2.app[3].payload = intWithUnit(uniform(1500Byte, 1500Byte))
```

```
**.nodo2.app[3].sendInterval = uniform(200us,500us)
**.nodo2.app[3].destination_gates = "phy[0]"
```

Código 1.6 Archivo INI del nodo 2

Dentro de la configuración de switch primero se enlazan los controladores a las tarjetas físicas del switch. Luego, para el VL 100 se define la ventana de tiempo, el destino, el VLID y la frecuencia. Para el VL 101 se define la frecuencia, el destino, la prioridad y el VLID.

```
[General]
network = RED

**.switch1.phy[0].inControl.ct_incomings = "vl_100_ctc"
**.switch1.phy[1].inControl.ct_incomings = "vl_101_ctc"
**.switch1.phy[1].shaper.tt_buffers = "vl_100"

**.switch1.vl_100_ctc.receive_window_start = sec_to_tick(995.120us)
**.switch1.vl_100_ctc.receive_window_end = sec_to_tick(1015.120us)
**.switch1.vl_100_ctc.permanence_pit = sec_to_tick(1015.120us)
**.switch1.vl_100.destination_gates = "phy[1].TTin"
**.switch1.vl_100.ct_id = 100
**.switch1.vl_100.sendWindowStart = sec_to_tick(1024.960us)

**.switch1.vl_101_ctc.bag = sec_to_tick(880us)
**.switch1.vl_101.destination_gates = "phy[0].RCin"
**.switch1.vl_101.bag = sec_to_tick(900us)
**.switch1.vl_101.priority = 0
**.switch1.vl_101.ct_id = 101
```

Código 1.7 Archivo INI del switch

Un último archivo INI es necesario para concretar especificaciones de la simulación. En este archivo, usualmente llamado *omnetpp.ini,* se indica el tiempo máximo de simulación, guardado de datos, plugins, la duración de un tic de reloj, parámetros de la sincronización entre relojes y finalmente, incluir los archivos INI creados anteriormente.

```
check-signals = true
sim-time-limit = 50s
tkenv-plugin-path = ../../../etc/plugins
**.vector-recording = true
**.scalar-recording = true

network = RED

# Global config
**.ct_marker = 0x03040506
**.ct_mask = 0xffffffff
```

```
# Scheduling parameters for all modules:
# Tick length 80ns
**.scheduler.tick = 80ns

# 62500 ticks result in cycle time of 5ms
**.scheduler.cycle_ticks = 62500tick

# Maximum clock drift of 2ps per tick
**.scheduler.max_drift = 2ps

# Maximum clock drift change of 0.1ps per cycle
**.scheduler.drift_change =  uniform(-10ps,10ps)

# precission of synchronisation
**.precission = 500ns

include nodo2.ini
include nodo1.ini
include switch.ini
```

Código 1.8 Archivo omnet.ini

1.3.3 Simulación

Una vez terminada la creación de la red, el entorno de trabajo tendrá una estructura similar a la de la fig. 1.7.

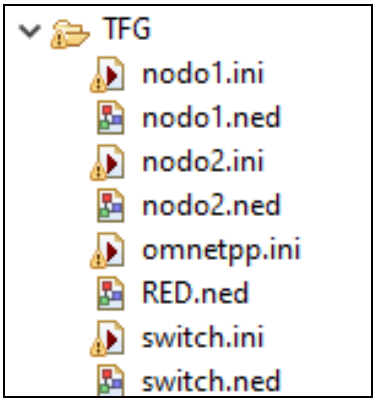


Fig. 1.7 Archivos de la red creada

Para empezar la simulación hay que clicar en la barra de herramientas superior, en la flecha negra a la derecha del *play* y selecionar *Run configurations…*

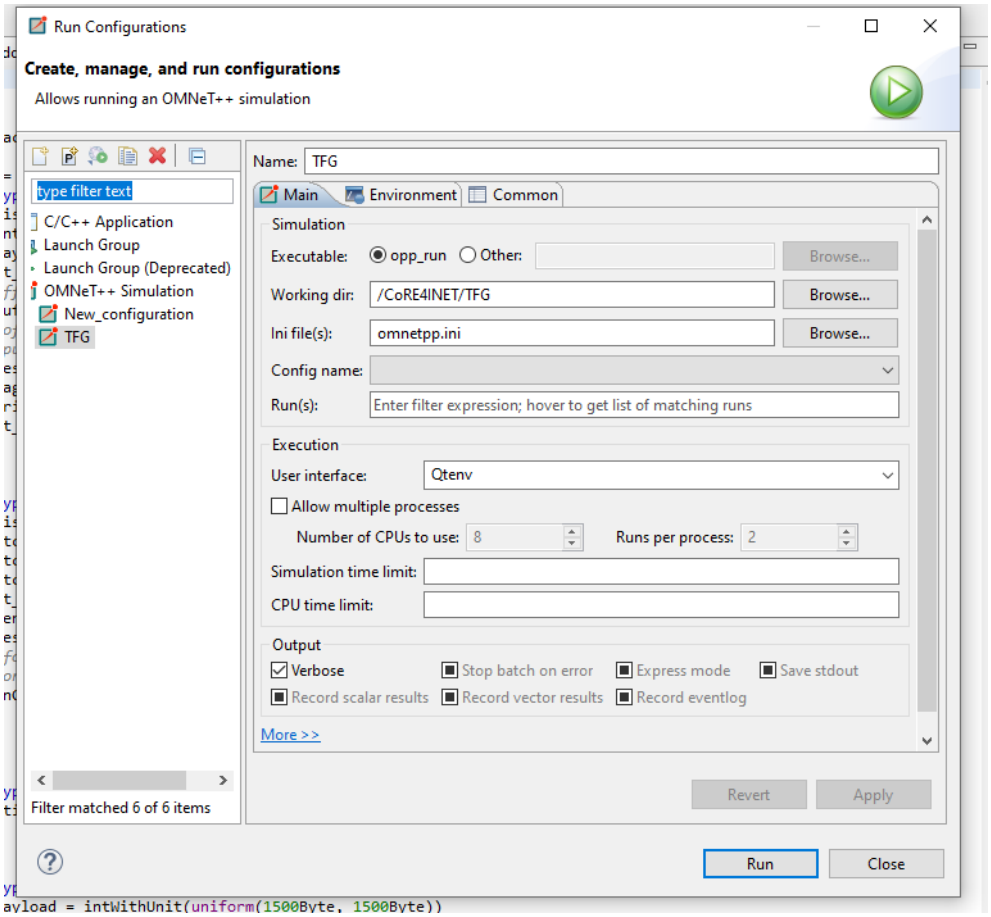


Fig. 1.8 Ventana *Run Configurations*

Dentro de esta ventana hay que fijarse si esta bien selecionado el directorio del entorno de trabajo y si esta selecionado el fichero INI que configura la simulación. En el apartado de ejecución, podemos selectionar el tipo de interfaz de usuario. Selecionando *Qtenv,* durante la simulación se abrira una representación gráfica de la red, muy interesante para ver como se comporta. Si se seleciona *Cmdenv* se correrá la simulación sin representarla gráficamente. Este modo es interesante para recopilar datos y analizarlos posteriormente. Esta ventana *Run Configurations* también permite limitar la simulación con el tiempo de simulación, los segundos simulados internamente, o el tiempo de CPU, que es cuanto tiempo está el ordenador haciendo los calculos. Estos valores tambien se pueden especificar dentro del archivo *omnetpp.ini*.

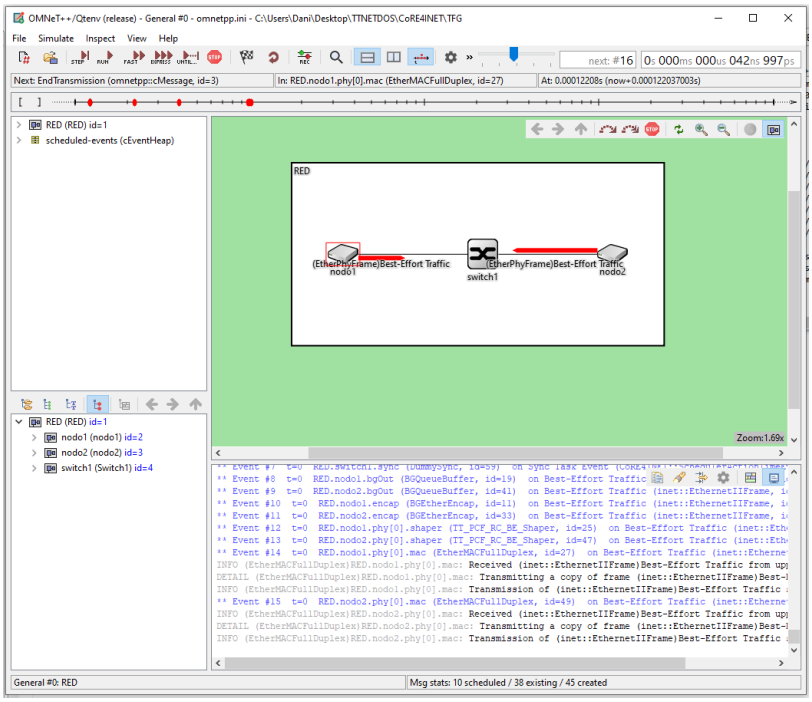


Fig. 1.9 Simulación con la interfaz Qtenv.

1.3.4 Resultados

Una vez hecha una simulación, aparecerá en el entorno de trabajo, si en el archivo *omnetpp.ini* está especificado que se guarden resultados, una carpeta llamada *results*. Dentro de ella habrá tres archivos, uno .sca para los escalares, uno .vec para vectores, y uno .vci que utiliza OMNeT++ internamente. Si se intenta abrir el archivo de vectores o escalares, el programa nos pedirá crear un archivo de análisis (.anf). Una vez creado, lo abrimos y entramos en la ventana *Browse Data* situada en la parte inferior de la ventana donde podremos ver todos los datos recogidos durante la simulación (ver fig. 1.11).

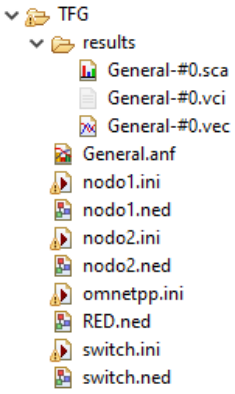
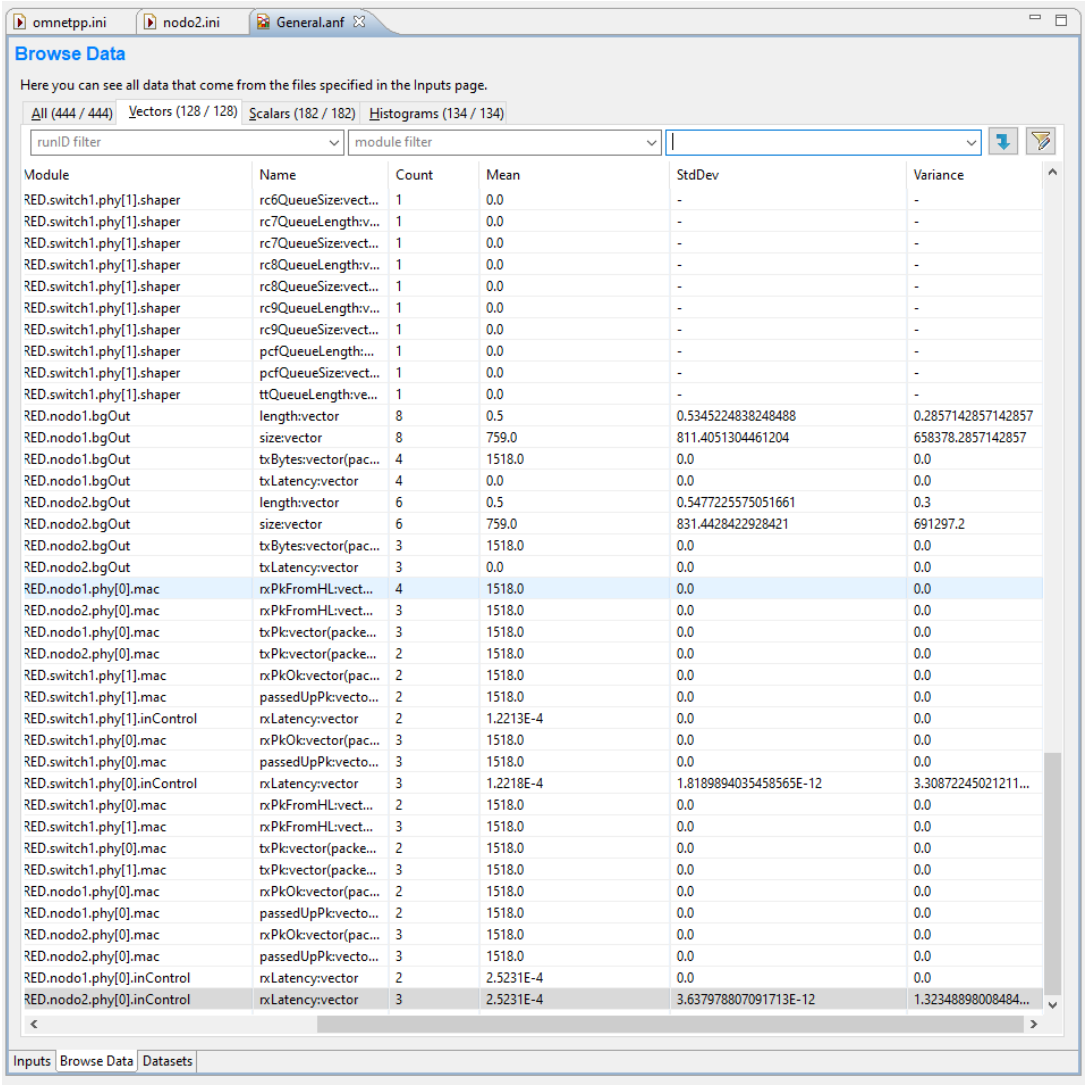


Fig. 1.10 Carpeta con los resultados



Module	Name	Count	Mean	StdDev	Variance
RED.switch1.phy[1].shaper	rc6QueueSize:vect...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc7QueueLength:v...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc7QueueSize:vect...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc8QueueLength:v...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc8QueueSize:vect...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc9QueueLength:v...	1	0.0	-	-
RED.switch1.phy[1].shaper	rc9QueueSize:vect...	1	0.0	-	-
RED.switch1.phy[1].shaper	pcfQueueLength:...	1	0.0	-	-
RED.switch1.phy[1].shaper	pcfQueueSize:vect...	1	0.0	-	-
RED.switch1.phy[1].shaper	ttQueueLength:ve...	1	0.0	-	-
RED.nodo1.bgOut	length:vector	8	0.5	0.5345224838248488	0.2857142857142857
RED.nodo1.bgOut	size:vector	8	759.0	811.4051304461204	658378.2857142857
RED.nodo1.bgOut	txBytes:vector(pac...	4	1518.0	0.0	0.0
RED.nodo1.bgOut	txLatency:vector	4	0.0	0.0	0.0
RED.nodo2.bgOut	length:vector	6	0.5	0.5477225575051661	0.3
RED.nodo2.bgOut	size:vector	6	759.0	831.4428422928421	691297.2
RED.nodo2.bgOut	txBytes:vector(pac...	3	1518.0	0.0	0.0
RED.nodo2.bgOut	txLatency:vector	3	0.0	0.0	0.0
RED.nodo1.phy[0].mac	rxPkFromHL:vect...	4	1518.0	0.0	0.0
RED.nodo2.phy[0].mac	rxPkFromHL:vect...	3	1518.0	0.0	0.0
RED.nodo1.phy[0].mac	txPk:vector(packe...	3	1518.0	0.0	0.0
RED.nodo2.phy[0].mac	txPk:vector(packe...	2	1518.0	0.0	0.0
RED.switch1.phy[1].mac	rxPkOk:vector(pac...	2	1518.0	0.0	0.0
RED.switch1.phy[1].mac	passedUpPk:vecto...	2	1518.0	0.0	0.0
RED.switch1.phy[1].inControl	rxLatency:vector	2	1.2213E-4	0.0	0.0
RED.switch1.phy[0].mac	rxPkOk:vector(pac...	3	1518.0	0.0	0.0
RED.switch1.phy[0].mac	passedUpPk:vecto...	3	1518.0	0.0	0.0
RED.switch1.phy[0].inControl	rxLatency:vector	3	1.2218E-4	1.8189894035458565E-12	3.308722450212 11...
RED.switch1.phy[0].mac	rxPkFromHL:vect...	2	1518.0	0.0	0.0
RED.switch1.phy[1].mac	rxPkFromHL:vect...	3	1518.0	0.0	0.0
RED.switch1.phy[0].mac	txPk:vector(packe...	2	1518.0	0.0	0.0
RED.switch1.phy[1].mac	txPk:vector(packe...	3	1518.0	0.0	0.0
RED.nodo1.phy[0].mac	rxPkOk:vector(pac...	2	1518.0	0.0	0.0
RED.nodo1.phy[0].mac	passedUpPk:vecto...	2	1518.0	0.0	0.0
RED.nodo2.phy[0].mac	rxPkOk:vector(pac...	3	1518.0	0.0	0.0
RED.nodo2.phy[0].mac	passedUpPk:vecto...	3	1518.0	0.0	0.0
RED.nodo1.phy[0].inControl	rxLatency:vector	2	2.5231E-4	0.0	0.0
RED.nodo2.phy[0].inControl	rxLatency:vector	3	2.5231E-4	3.637978807091713E-12	1.32348898008484...

Fig. 1.11 Ventana *Browse Data* del archivo de análisis

Las funcionalidades interesantes de los archivos de análisis son poder filtrar, agrupar datos en sets, plotear vectores individualmente o en grupo, hacer histogramas con escalares, etc. Dentro de la tabla de vectores se pueden añadir columnas con el valor medio, la deviación estándar, la varianza, máximos y mínimos, etc. También se puede exportar fácilmente los datos para comparar diferentes configuraciones o redes, o para trabajar desde Excel. Por ejemplo, en el siguiente gráfico (ver fig. 1.12) se están representando los vectores de latencia de el VL 100 (TT) y del VL 101 (RC).

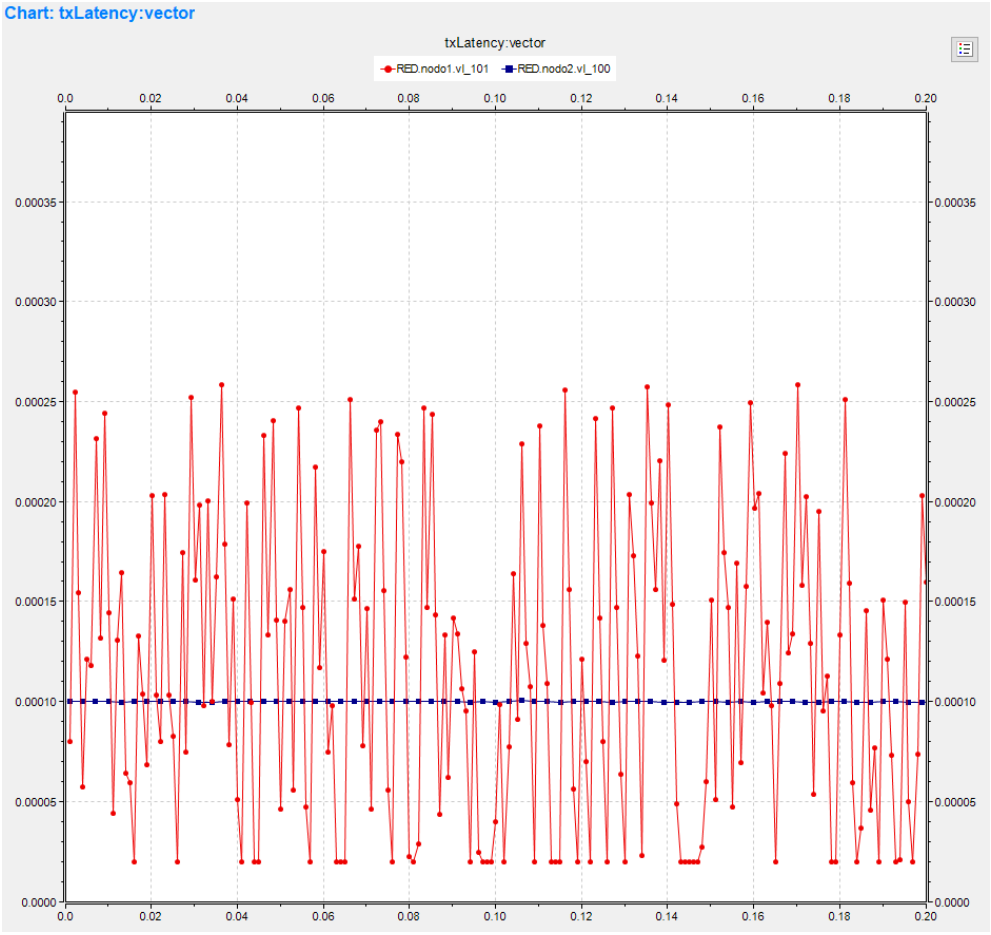


Fig. 1.12 Gráfico de latencia

1.3.5 Red *Small Network* de CoRE4INET

El paquete CoRE4INET viene con una serie de redes TTEthernet de ejemplo. Estas redes pueden ser aprovechadas para analizar las características de una red TTEthernet. Por ejemplo, esta primera red (ver fig. 1.13), que se encuentra en el siguiente directorio (CoRE4INET\examples\AS6802\small_network).

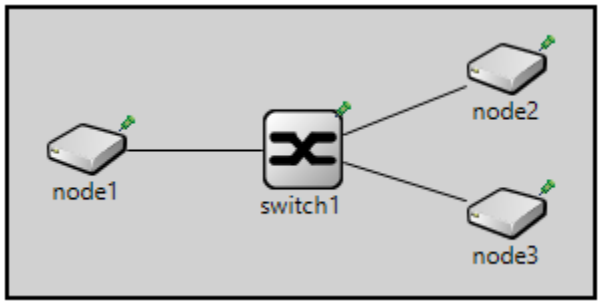


Fig. 1.13 Red *Small Network*

La red *Small Network* consta de 3 nodos y un switch. Hay un virtual link TT (VL100) en el cual el nodo 1 envía paquetes al nodo 2 y 3, un virtual link RC (VL101) en el cual el nodo 2 envía datos al nodo 3, y un último virtual link TT (VL102) donde el nodo 1 envía paquetes al 2. También se usará el modo *crosstraffic* donde se aplicará tráfico BE fuera de los *virtual links*. El nodo 1 está conectado por un cable de 20 metros al switch, el nodo 2 por uno de 10 metros y el 3 por uno de 5 metros, todas las conexiones son con 100 Mbps Ethernet. Se obtendrán datos de una simulación de 2 minutos de tiempo de CPU en *Cmdenv*, equivalente a 37,2 segundos de tiempo de simulación.

Es una red sencilla para obtener valores limpios y fáciles de analizar de una red TTEthernet.

1.3.5.1 Latencia:

La latencia en una red es un término utilizado para indicar cualquier tipo de retraso que ocurre en transmisiones de datos en una red, incluyendo el tiempo que gasta el mensaje para propagarse por el medio físico, como retrasos surgidos por compartir el canal con otros dispositivos.

1.3.5.2 Jitter:

Jitter o fluctuación del retardo es la variación en tiempo de entre paquetes de datos. Es una disrupción en la secuencia de enviado de paquetes ideal. En el siguiente gráfico están representadas las latencias entre transmisor y receptor de cada VL durante los primeros 30 milisegundos de la simulación.

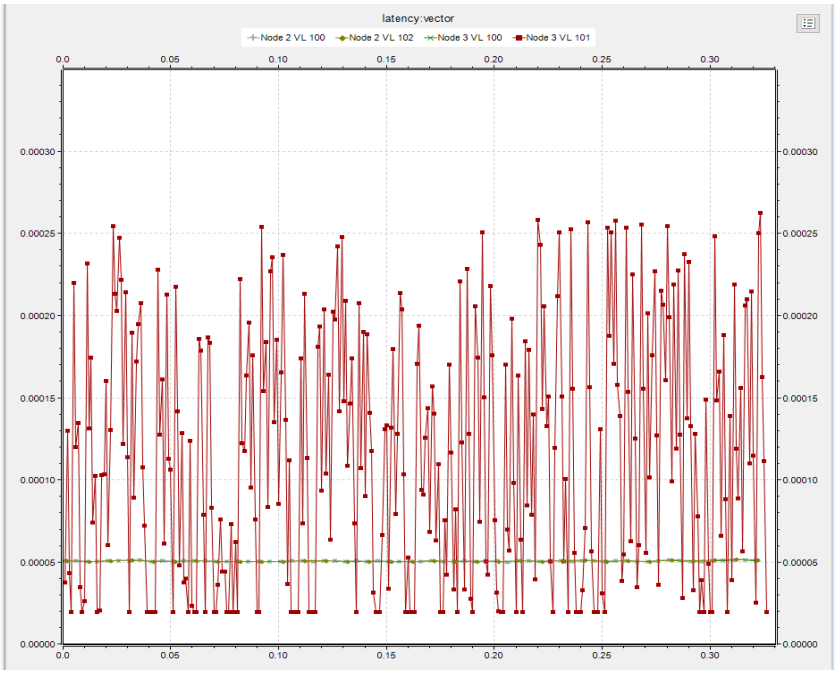


Fig. 1.14 Gráfico de latencias por VL

En la fig. 1.14 se observa a primera vista como la VL que utiliza tráfico RC tiene una latencia media y una varianza de ella (jitter) mayor que las VL de tráfico TT, que han obtenido un valor de latencia menor y un jitter casi nulo. También es visible que la latencia mínima del tráfico RC es menor que la del tráfico TT, por lo que, en un canal poco congestionado, la comunicación RC podría llegar a tener menos latencia que la comunicación TT. Esto se debe a que el tráfico TT tiene una latencia generada en el switch surgida de la espera entre la recepción del mensaje y el inicio del slot temporal de transmisión. Un diseño óptimo de la tabla de slots temporales disminuye esta latencia.

Tabla 1.1 Latencia, jitter y datos recibidos de la red *Small Network*

Sistema terminal	Latencia	Jitter	Datos recibidos
Nodo 2 VL 100 (TT)	50.7 µs	261 ns	465 kbits
Nodo 2 VL 102 (TT)	50.8 µs	346 ns	232.5 kbits
Nodo 3 VL 100 (TT)	50.7 µs	261 ns	465 kbits
Nodo 3 VL 101 (RC)	114.6 µs	76.2 µs	2.27 Mbits
Nodo 1 BE	252.8 µs	102.3 µs	153.82 Mbits
Nodo 2 BE	257.3 µs	134.3 µs	153.84 Mbits
Nodo 3 BE	253.9 µs	203.1 µs	153.88 Mbits

Estos resultados, obtenidos de los archivos de análisis de OMNeT++ (ver fig. 1.11) muestran la importancia de escoger cada tipo de mensaje para la aplicación adecuada, según el requerimiento de seguridad en la transmisión o cantidad de datos. También hay que tener en cuenta que esta red solo utiliza 3 VL, y el tráfico BE está configurado para ocupar todo el ancho de banda disponible (para testear la red en condiciones críticas) provocando esta gran diferencia de cantidad de datos recibidos.

1.3.5.3 Frame del mensaje: cabeceras y parámetros

Los mensajes TTEthernet tienen los siguientes parámetros: MAC origen, MAC destino, Ethertype, Payload y CRC. TTEthernet no utiliza direcciones origen y destino IP, solo utiliza direcciones MAC para el origen, y el VLID para el destino. La MAC de destino en TTE conserva para mantener compatibilidad con Ethernet y TTE la divide en dos partes: los primeros bits reflejan la dirección MAC local y los 16 últimos el VLID. El Ethertype está compuesto por dos bytes e indica con que protocolo se está transmitiendo en el mensaje. *Payload* son los bytes reservados para los datos (de 46 a 1500 bytes). El CRC es el *Cyclic redundancy check,* un sistema de detección y corrección de errores.

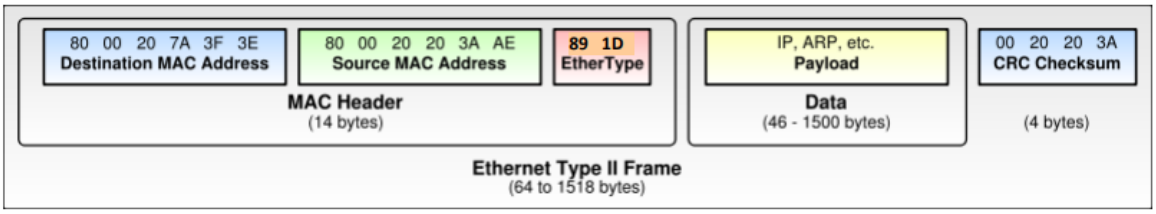


Fig. 1.15 Cabeceras del mensaje TTE

En OMNeT++ podemos pausar la simulación gráfica y hacer clic en un mensaje para abrir una ventana con su información. Dentro de esa ventana se encuentran las cabeceras de TTEthernet que utiliza el simulador y compararlas con las cabeceras que debería tener un mensaje TTE real.

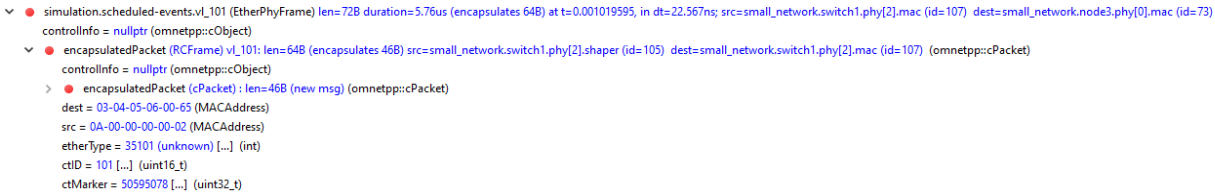


Fig. 1.16 Cabeceras del mensaje TTE dentro de OMNeT++

Además de las cabeceras TTEthernet, en la ventana del mensaje nos encontramos con información extra, como el tiempo de creación del mensaje, el de llegada, el tiempo que dura la transmisión, longitud en bits, etc. Con estas ventanas podemos comparar diferentes tipos de mensaje y detectar diferencias, como que cada mensaje porta su VLID correspondiente, o ninguno si es un mensaje BE.

1.3.5.4 Funcionamiento de los nodos y switches

En este apartado se explica cómo están diseñados interiormente los nodos y el switch que utiliza la extensión CoRE4INET, y qué camino sigue el mensaje desde su creación hasta la recepción. Todos estos pasos quedan registrados como eventos en la consola dentro de la simulación gráfica (ver fig. 1.17).

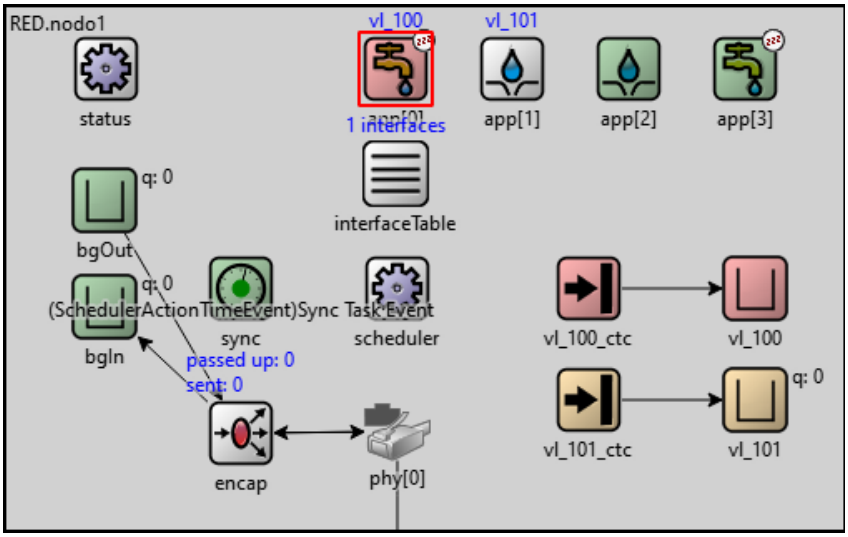


Fig. 1.17 Estructura del nodo 1

```
** Event #1  t=0  small_network.node1.scheduler.timer (Timer, id=29)  on selfmsg Scheduler Message (omnetpp::cMessage, id=0)
** Event #2  t=0  small_network.node1.app[0] (TTTrafficSourceApp, id=17)  on selfmsg Start Message (omnetpp::cMessage, id=9)
** Event #3  t=0  small_network.node1.app[1] (TTTrafficSourceApp, id=18)  on selfmsg Start Message (omnetpp::cMessage, id=10)
** Event #8  t=0  small_network.node1.sync (DummySync, id=15)  on Sync Task Event (CoRE4INET::SchedulerActionTimeEvent, id=33)
** Event #12  t=0.00098  small_network.node1.scheduler.timer (Timer, id=29)  on selfmsg Scheduler Message (omnetpp::cMessage, id=0)
** Event #13  t=0.00098  small_network.node1.app[0] (TTTrafficSourceApp, id=17)  on API Scheduler Task Event (CoRE4INET::SchedulerActionTimeEvent, id=56)
** Event #14  t=0.00098  small_network.node1.vl_100_ctc (TTIncoming, id=22)  on vl_100 (CoRE4INET::TTFrame, id=60)
** Event #15  t=0.00099  small_network.node1.scheduler.timer (Timer, id=29)  on selfmsg Scheduler Message (omnetpp::cMessage, id=0)
** Event #16  t=0.00099  small_network.node1.vl_100_ctc (TTIncoming, id=22)  on PIT Event (CoRE4INET::SchedulerActionTimeEvent, id=63)
** Event #17  t=0.00099  small_network.node1.vl_100 (TTDoubleBuffer, id=23)  on vl_100 (CoRE4INET::TTFrame, id=60)
** Event #21  t=0.001  small_network.node1.scheduler.timer (Timer, id=29)  on selfmsg Scheduler Message (omnetpp::cMessage, id=0)
** Event #26  t=0.001  small_network.node1.scheduler.period[1] (Period, id=32)  on Period New Cycle Event (CoRE4INET::SchedulerActionTimeEvent, id=8)
** Event #27  t=0.001  small_network.node1.vl_100 (TTDoubleBuffer, id=23)  on TTBuffer Scheduler Event (CoRE4INET::SchedulerActionTimeEvent, id=36)
** Event #29  t=0.001  small_network.node1.phy[0].shaper (TT_PCF_RC_BE_Shaper, id=26)  on vl_100 (CoRE4INET::TTFrame, id=76)
** Event #31  t=0.001  small_network.node1.phy[0].mac (EtherMACFullDuplex, id=28)  on vl_100 (CoRE4INET::TTFrame, id=76)
INFO (EtherMACFullDuplex)small_network.node1.phy[0].mac: Received (CoRE4INET::TTFrame)vl_100 from upper layer.
DETAIL (EtherMACFullDuplex)small_network.node1.phy[0].mac: Transmitting a copy of frame (CoRE4INET::TTFrame)vl_100
INFO (EtherMACFullDuplex)small_network.node1.phy[0].mac: Transmission of (CoRE4INET::TTFrame)vl_100 started.
```

Fig. 1.18 Lista de eventos del nodo 1 al iniciar la simulación

El *Scheduler* tiene guardados los tiempos de transmisión de la comunicación TT. *App,* la aplicación, genera un mensaje y lo envía al *ctc (*controllador*)* correspondiente de su VL, el módulo que valida los mensajes TT. El *ctc* comprueba los tiempos con el *scheduler* y manda el mensaje a un buffer. Cuando es el momento de enviar el mensaje, pasa del buffer a *encap*, el encapsulador que crea las cabeceras, y después a *phy,* el puerto ethernet, donde se enviará a la red. El puerto ethernet prioriza los mensajes en el siguiente orden: TT, RC, BE.

El dispositivo *status* controla el estado de la red (up, down, etc.) para los otros módulos. *BgIn* y *bgOut* son buffers que se encargan del *background traffic,* en TTEthernet, el tráfico BE. *Sync* es el módulo que se encarga de la sincronización entre dispositivos y las correcciones de reloj. *InterfaceTable* almacena la información de las interfaces de la red.

Una vez el mensaje sale del transmisor llega al switch por su puerto ethernet y actúa *beswitch*. Este módulo controla la asignación entre puertos y direcciones

MAC, almacenada en la *macTable*. Lee las cabeceras del mensaje y lo envía al buffer correspondiente. Cuando el *scheduler* avise que es el momento de la transmisión, se envía el mensaje del buffer a los puertos asignados al VL y se envía a los receptores. La *interface table* se encarga de registrar dinámicamente las interfaces.

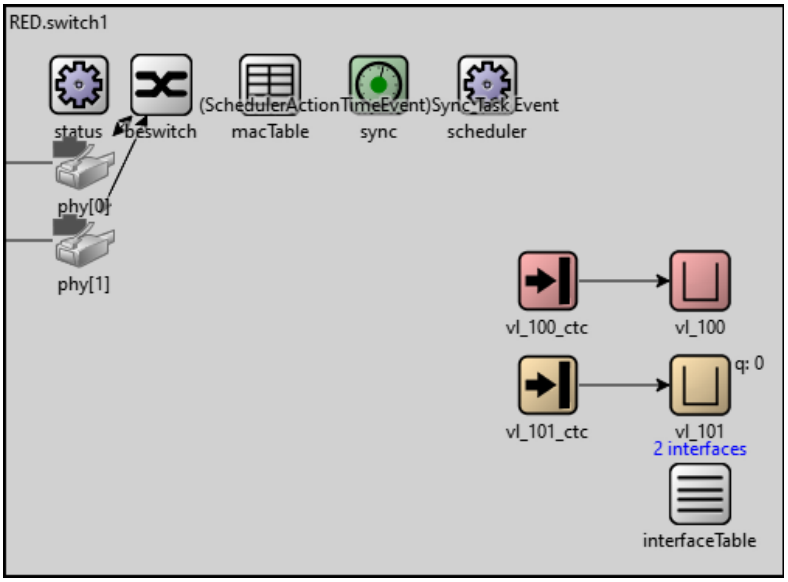


Fig. 1.19 Estructura del Switch

Una vez en el receptor, el mensaje pasa de la tarjeta física, pasa por el controlador (si el mensaje es TT o RC) hacia el buffer correspondiente, de VL o *back ground*, y finalmente a la aplicación.

1.3.5.5 Reloj, sincronización

Los equipos TTEthernet son capaces de detectar inconsistencias en la sincronización de reloj y comunicarlas al resto de los sistemas. Estas inconsistencias tienen que ser reparadas cooperativamente, ya que las redes TTEthernet no tienen un único *clock master.* Si un equipo en particular no es capaz de sincronizarse y el suficiente número de equipos ha detectado ese fallo, ese equipo es excluido de la comunicación.

El algoritmo de sincronización asigna a los dispositivos de la red uno de tres roles, máster de sincronización (SM), máster de compresión (CM) y cliente de sincronización (SC). Generalmente los switch son CM y los equipos terminales SM. SM y CM intercambian PCF (*protocol control frames*) entre ellos para comprobar el estado de la red. Estos PCF son mensajes de control que no envían información útil, pero son necesarios para garantizar la sincronización de los dispositivos de la red.

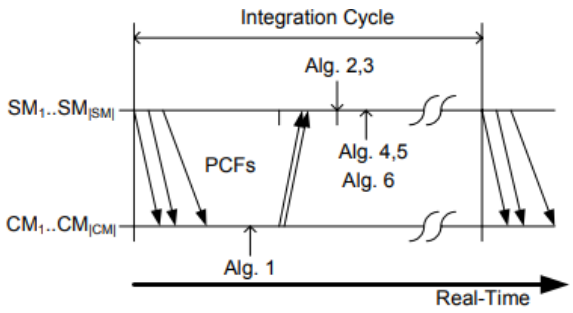


Fig. 1.20 Metodo de sincronización de reloj

Esta ilustración muestra el algoritmo de sincronización en dos pasos. Primero, el SM envía un PCF al CM. El CM calcula el estado del reloj local del SM y ejecuta la primera función convergente, conocida como *compression function* (Alg. 1 de la Fig. 1.20). El resultado es enviado al SM en un nuevo PCF. El segundo paso es el SM leyendo el PCF recibido y ejecutando la segunda función convergente (Alg. 2,3) para corregir el reloj. Después de esta corrección se aplican el algoritmo de análisis (Alg. 4,5) y el algoritmo de corrección de tasa (Alg 6). Los dispositivos SC se sincronizan pasivamente leyendo el intercambio de mensajes descrito entre CM y SM.

El algoritmo de análisis es el encargado de expulsar del algoritmo de sincronización a un equipo que no pare de generar errores de sincronización y el algoritmo de corrección de tasa se encarga de almacenar las correcciones antiguas de reloj, para buscar una tendencia en el error y así disminuirlo con una corrección preventiva.

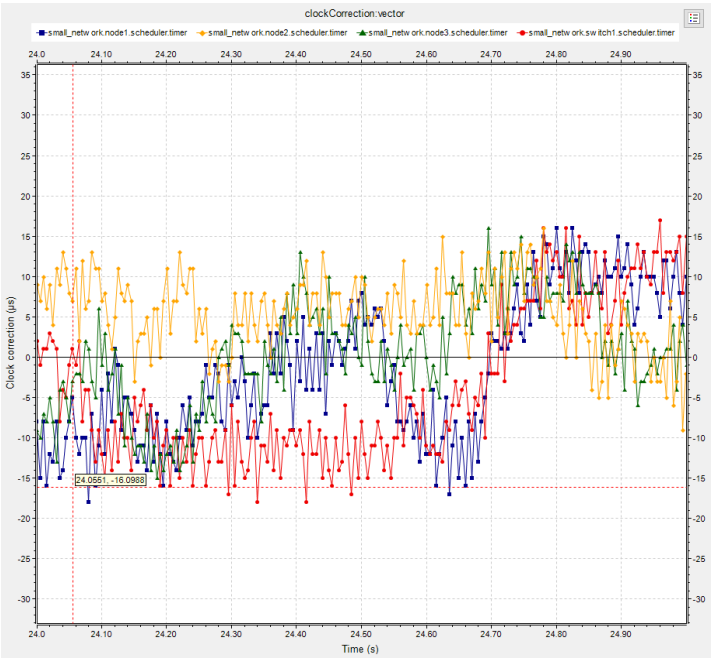


Fig. 1.21 Correcciones de reloj durante la simulación

En este gráfico están registradas las correcciones de error producidas durante el segundo 24 de la primera simulación por los 3 nodos i el switch. Se observa que cada nodo hace decenas de correcciones por segundo durante el funcionamiento normal de la red. Hay una tendencia a mantenerse cerca de 0, lo que equivale a una sincronización ideal. También se puede sacar tendencias de comportamiento, por ejemplo, el nodo 2, representado en amarillo, mayormente tiene correcciones positivas, por lo que su reloj tiende a ralentizarse.

1.3.6 Red *Large Network* de CoRE4INET

Esta segunda red de ejemplo que provee CoRE4INET (situada en el fichero *CoRE4INET/examples/IEEE8021Qci/large_network*) es ya más compleja, cuenta ya con 3 switches y 10 nodos. Hay comunicación TT entre los nodos 3, 4, 5, 6, 7 y 9, el nodo 10 transmite broadcast BE con respuesta. Esta simulación cuenta también con una comunicación AVB, otro estándar dentro de IEEE 802.1 que permite una transmisión de video utilizando tráfico RC. Este estándar es compatible con TTEthernet, como lo es AFDX que también utiliza tráfico RC, y la simulación permite su desactivación. En esta simulación también se le aplica el estándar IEEE 802.1Qci llamado *Per-Stream Filtering and Policing*. Este estándar se encarga de filtrar en switches para detectar y eliminar paquetes disruptivos de algún posible nodo corrupto.

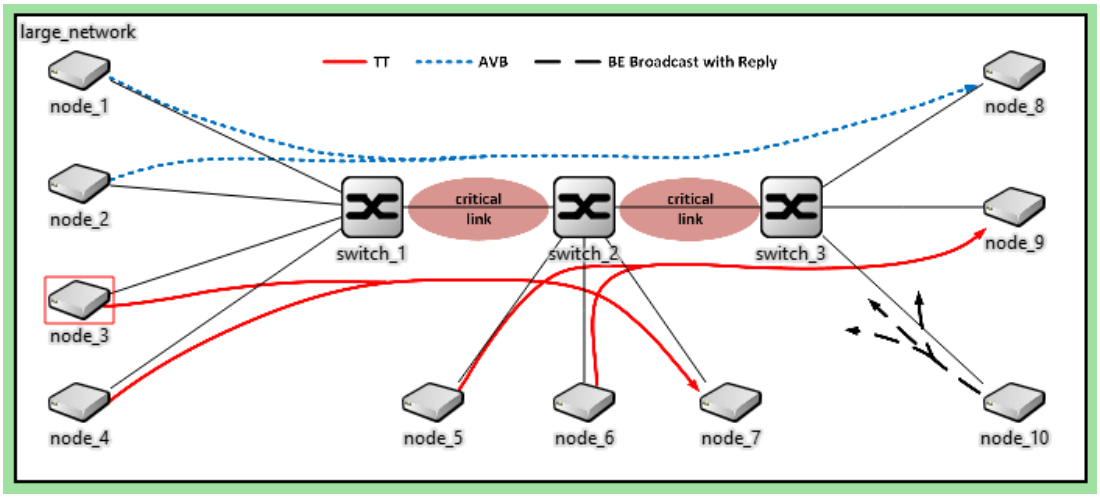


Fig. 1.22 Esquema de la red *Large Network*

1.3.6.1 Simulación y resultados

La simulación de esta red se ha hecho con tres configuraciones diferentes:

- Configuración 1: Sin tráfico AVB
- Configuración 2: Con tráfico AVB
- Configuración 3: Con tráfico AVB y con un tiempo entre mensajes TT mayor (el doble)

1.3.6.2 Latencia

Los resultados de latencia, la media entre todos los dispositivos que comparten un mismo tipo de tráfico (TT, RC o BE), obtenidos muestran comportamientos interesantes de la red. Primero, comparando la configuración 1 y 2, podemos observar como la adición del tráfico AVB no afecta a la latencia del tráfico TT. Esto se debe a que los mensajes TT tienen prioridad absoluta en su slot temporal asignado. Por el contrario, podemos observar un augmento de latencia en el tráfico BE al tener que compartir su ancho de banda con el tráfico AVB.

Comparando la configuración 2 y 3, se observa como la disminución de slots TT hace augmentar la latencia de este tráfico, al haber intervalos de tiempo más grandes entre mensajes TT, desde la generación del mensaje hasta el envío pasa más tiempo, por lo que augmenta la latencia. Esto se debe que aunque la comunicación sea *time triggered,* la aplicación que genera el mensaje no lo es, por ejemplo, una decisión del piloto de cambio de altitud, el mensaje lo genera la aplicación con el input del piloto y debe esperar a su slot temporal para enviarse, generando una latencia en el sistema. Esta latencia es inversamente proporcional a la frecuencia de slots temporales.

A su vez, se disminuye la latencia del tráfico BE y AVB, al tener más ancho de banda disponible.

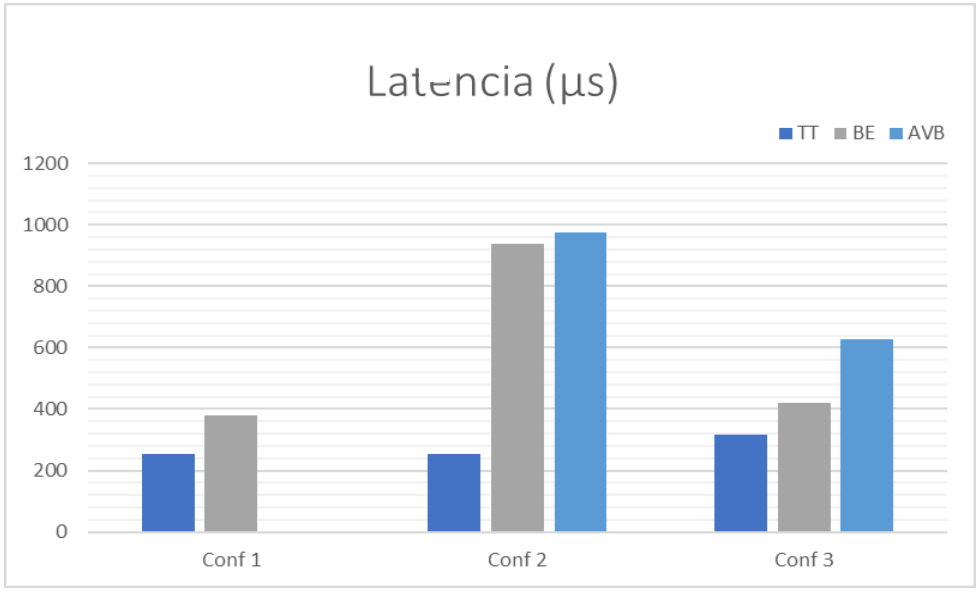


Fig. 1.23 Comparación de latencias

1.3.6.3 Jitter

El jitter obtenido durante la simulación del tráfico TT para cada configuración es 142 ns, 135 ns, y 131 ns respectivamente, al ser valores tan pequeños en comparación con el resto, no se pueden apreciar bien en la gráfica. Un jitter bajo es característico de las comunicaciones basadas en tiempo y podemos comprobar como el efecto de añadir tráfico a la red o disminuir la frecuencia de transmisión de mensajes TT no tiene un efecto notable en el jitter. Por el otro lado, el jitter en el tráfico BE sí augmenta al disminuir el ancho de banda disponible, ya que, en una comunicación basada en eventos, cuanto menos ancho de banda disponible haya, más fácil es que se sature el canal.

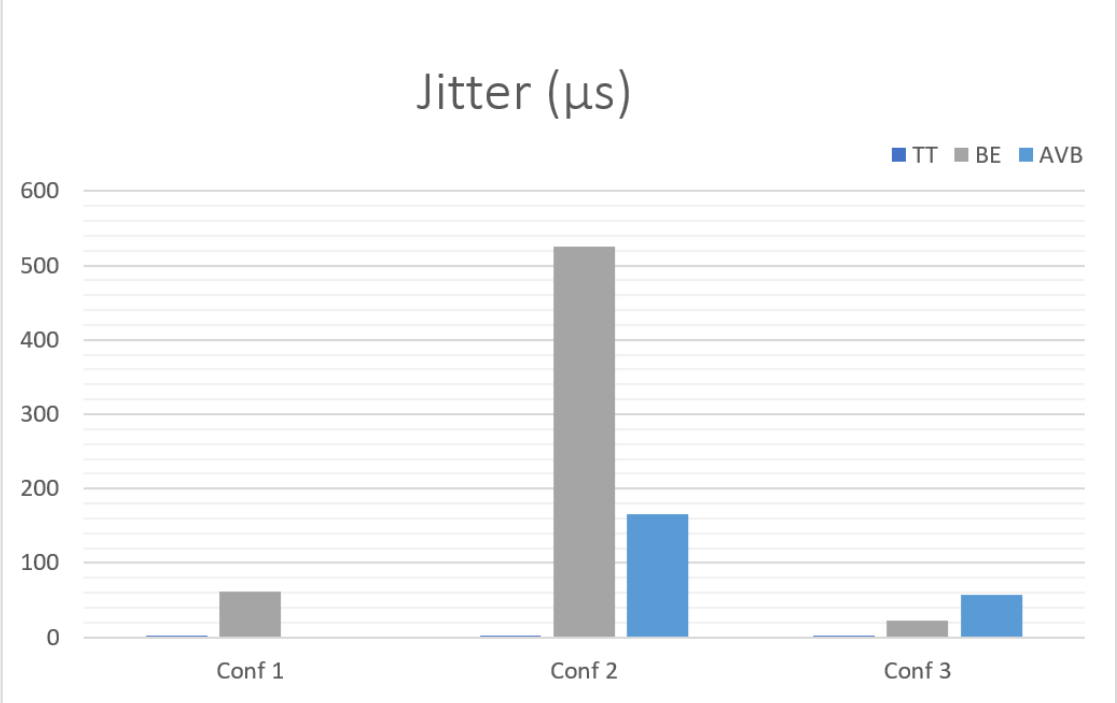


Fig. 1.24 Comparación de jitters

1.3.6.4 Correcciones de reloj

Para la corrección de reloj se ha calculado la media absoluta por configuración, ya que una corrección de reloj puede ser positiva o negativa. La diferencia entre configuraciones probablemente es negligible, al ser variaciones de ~2%.

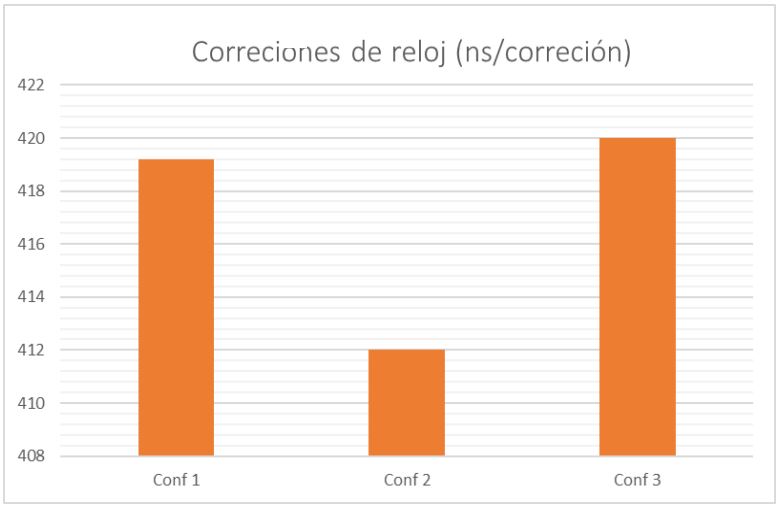


Fig. 1.25 Comparación de media de tiempo corregido por corrección

1.3.6.5 Bit rate

Los resultados de bit rate muestran como para la simulación 1, donde no hay tráfico AVB, los switches tienen que trabajar con menos carga. Para las simulaciones 2 y 3, el tráfico es prácticamente idéntico, con diferencias del orden de 10 kbps. Este resultado puede ser consecuencia de trabajar al límite de capacidad de la red, el flujo de bits no varía al disminuir la frecuencia TT, pero sí las latencias de BE y AVB.

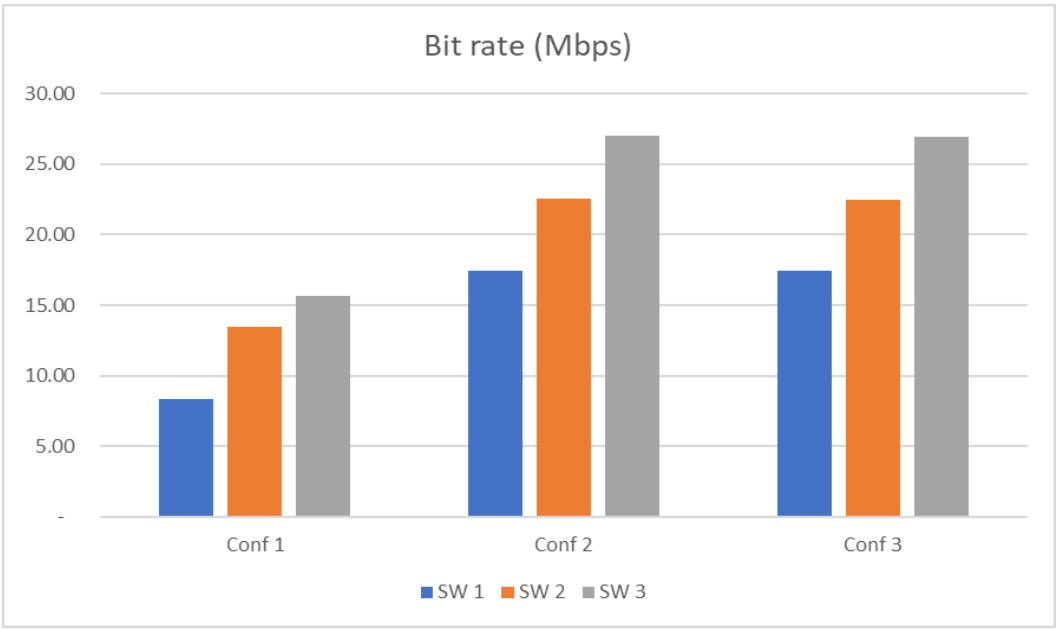


Fig. 1.26 Comparación de Bit rate en switches

1.4 Ejemplos y ejercicios académicos

Para empezar, se podría utilizar los ejemplos que ya trae el paquete CoRE4INET, como los utilizados en los apartados 2.3.5 y 2.3.6. Con una red ya construida se pueden modificar valores para entender comportamientos de una red TTE:

- Cambiar VLID: se puede cambiar el VLID en la aplicación que genera el mensaje, o en los controladores por los que tiene que pasar (en el emisor, switches y receptor). En el momento que el VLID del mensaje no coincida con el VLID del controlador, el mensaje será tirado.

- Cambiar *time slots* en los mensajes TT: se puede cambiar cuando se envía un mensaje TT y cuando se espera recibirlo. En el caso que el controlador detecte un mensaje TT fuera de tiempo, este será eliminado. Dentro de esta prueba también se puede aumentar la longitud del cable utilizado para las conexiones, para aumentar el tiempo de propagación.

- Cambiar tasas de transmisión y carga de mensajes: Variando estos dos valores cambiaran los resultados obtenidos al hacer una simulación en cuanto latencias, jitter, bit rates, utilización de buffers, etc.

- Cambiar el encaminamiento de los switches: se puede configurar el switch para que envíe mensajes a un nodo que no esté dentro del enlace VL, y así ver como este nodo elimina estos mensajes no deseados.

- Pausar la simulación y analizar los mensajes: pausando la simulación Qtenv, se puede clicar en los mensajes para ver una ventana con su información, como por ejemplo, su estructura, cabeceras y carga. Con esta información se puede prever su ruta en la red.

- Añadir un nodo a una red existente: se puede añadir un nodo a una red ya construida y reconfigurarla para que el nodo interactúe con los mensajes que se generan en la red y para transmitir los suyos. Es un ejercicio interesante para reconfigurar tablas de tiempos TT y tamaño de buffers.

Otro ejercicio sería la creación de una red TTE, con unas especificas características, y decidir para cada aplicación que tipo de mensaje utilizar, y con qué configuración.

CAPÍTULO 2. AFDX

2.1 Avionics Full-Duplex Switched Ethernet

Avionics Full-Duplex Switched Ethernet (AFDX), estandarizado como ARINC 664, es una especificación de redes de buses de comunicación aeronáutica determinista. Fue patentado por *Airbus* para su uso en el A380. En la actualidad, nuevos modelos de Airbus como el A350 también utilizan AFDX, e incluso *Boeing* utiliza AFDX en el B787 *Dreamliner*. Está construido bajo el estándar IEEE 802.3 utilizando tecnología Ethernet. La principal ventaja del uso de una tecnología tan utilizada como Ethernet es la reducción de costes en comparación de otras tecnologías propias de la aeronáutica como ARINC 429. Su topología en estrella ayuda a reducir el peso del avión al necesitar utilizar menos cables para las interconexiones aviónicas, en comparación con una red con una topología en bus.

El estándar comercial de Ethernet no cumple los requisitos aeronáuticos para una red de comunicación, y por ello AFDX extiende el estándar de Ethernet dotándole de un comportamiento determinista conseguido con anchos de banda dedicados (Virtual Links) y añadiéndole QoS (*Quality of Service*) para minimizar la probabilidad de error en la comunicación. Esta necesidad que QoS es necesaria en los aviones *fly-by-wire*.

El QoS necesario es conseguido con el uso una red de redundancia (ver fig. 2.1). En el despliegue con redundancia, cada nodo de la red deberá enviar el mensaje de manera simultánea por dos redes idénticas (normalmente denominadas redes A y B). Estas dos redes están en funcionamiento todo el tiempo, por lo que, durante el funcionamiento normal de la red, el receptor recibirá dos mensajes idénticos y descartará uno de ellos.

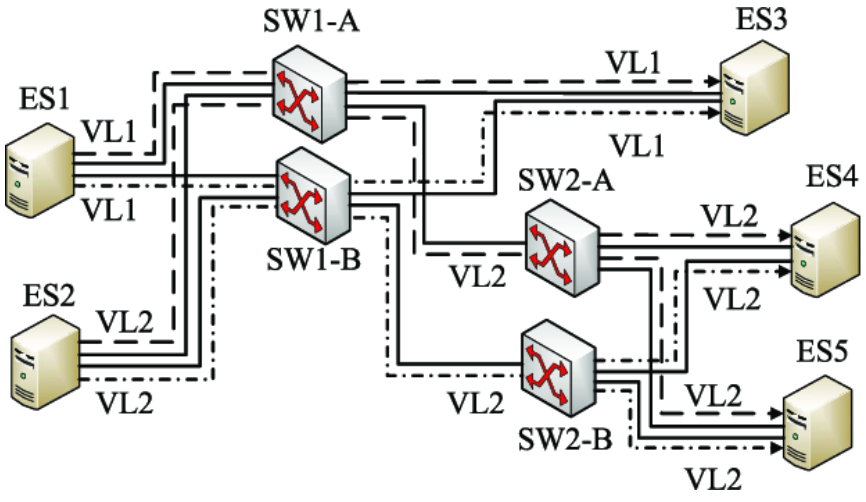


Fig. 2.1 Esquematización de una red AFDX con redundancia

AFDX soporta velocidades de hasta 1 Gbps, y podría augmentar con el avance de la tecnología Ethernet. Los nodos están conectados a los switches por una conexión *full-duplex*.

2.2 Simulador OMNeT++ y la extensión AFDX

Para el protocolo AFDX se seguirá usando la plataforma OMNeT++ y se le añadirá la extensión AFDX para simularla. Esta extensión solo implementa la capa MAC y su autor es Rudolf Hornig[7].

Para la instalación del paquete AFDX hay que descargarlo del repositorio de OMNeT++[2]. Una vez descargado y descomprimido, hay que depositarlo en el entorno de trabajo utilizado. Para instalarlo se tiene que entrar en el IDE, ir a *File, Import…;* seleccionar *General, Existing Projects into Workspace*, y seleccionar la carpeta de AFDX.

Dentro del paquete encontramos una red de ejemplo, formada por 4 nodos y 2 switches, uno para la red A y otro para la B, formando el despliegue con red de redundancia (ver fig. 2.2).

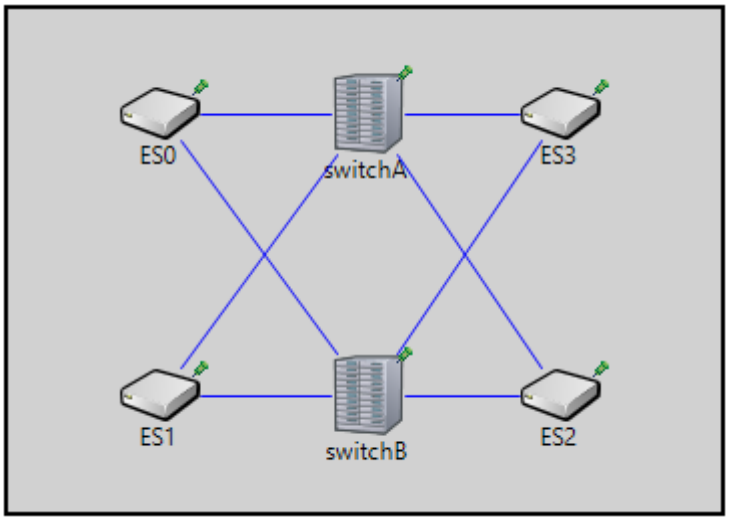


Fig. 2.2 Red del paquete AFDX

El autor especifica en la descripción del simulador que solo esta implementada la capa MAC, y esto contrae una serie de problemáticas. Una de ellas es que al no haber aplicaciones que controlen la creación y la recepción de mensajes, no se pueden medir ni latencia ni jitter. Otro inconveniente es que no se pueden crear VL, por lo que la tabla de encaminamiento del switch tiene que ser falseada para poder iniciar la simulación. El autor lo explica dentro del código, en */src/VLRouter.cc* (ver código 2.1). Lo que hace la simulación es generar en

cada mensaje un VLID aleatorio, entre el 0 y el 3, el switch lo lee y lo envía los nodos con el ID igual a VLID+1 o VLID -1.

```
void VLRouter::handleMessage(cMessage *msg)
{
    AFDXMessage *afdxMsg = check_and_cast<AFDXMessage *>(msg);

       int vlId = afdxMsg->getVirtualLinkId();
       // send a copy on each gate but the last, and the original on the last gate
       for (int i=0; i<gateSize("out"); i++)
       {
              // this is just a fake routing table that routes to the vlId-1 and
vlID+1 port
           // a correct routing algorithm should be implemented here possibly reading
the
           // rules from an external file
              if (vlId + 1 == i || vlId -1 == i)
                     send((cMessage *)afdxMsg->dup(), "out", i);
       }
       delete afdxMsg;
}
```

Código 2.1 Configuración de la tabla de encaminamiento

La estructura del nodo es la siguiente (ver fig. 2.3), el submódulo *trafficSource1* genera mensajes sin cabeceras de 128 bytes. El *regulatorLogic* se encarga de regular el BAG, el jitter máximo y la longitud permitida. El *redundacyController* duplica el mensaje y lo envía por el canal A y B. *TxQueue* es un buffer fifo (First in, first out). El submódulo *mac* direcciona los mensajes de las colas a la red, y los de la red al *integrityChecher,* el cual no tiene función definida en su código, el mensaje igual que entra sale. El *redundacyChecker* elimina uno de los mensajes de la redundancia y el *trafficSink* elimina el mensaje recibido sin guardar registro de él. Otra funcionalidad que tampoco tiene este simulador es que no hay velocidad de propagación del mensaje, el instante en el que un mensaje entra en un canal, sale de él.

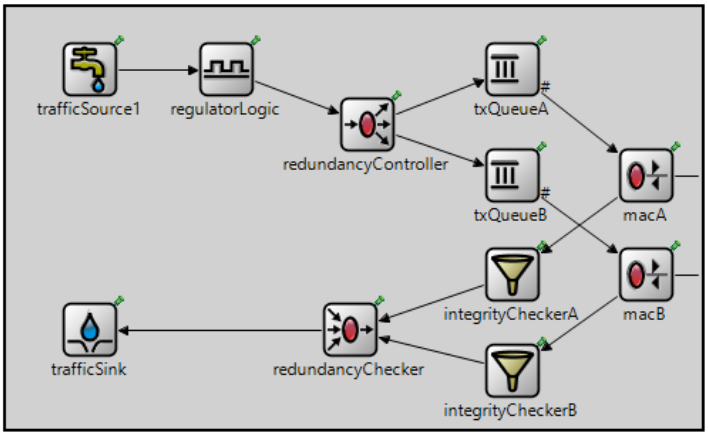


Fig. 2.3 Estructura de los nodos

Aun con estos problemas, esta simulación puede ser útil para entender el funcionamiento del despliegue de la red con redundancia. Con el IDE de OMNeT++ se puede ver cómo el nodo genera un único mensaje, lo duplica el controlador de redundancia y lo envía simultáneamente a las dos redes. Y en la recepción cómo recibe los dos mensajes a la vez, cómo el verificador de redundancia los compara y finalmente deja pasar uno y elimina el otro (ir al anexo 5.2 para ver la representación gráfica de este proceso).

Otro ejercicio a hacer con este simulador es poner a prueba la redundancia interrumpiendo el envio de mensajes en una red, por ejemplo desconectando el nodo 0 del switch A (ver fig. 2.4). Cada vez que el switch A y el nodo 0 se intenten comunicar se elimina el mensaje al no haber conexion, pero al tener la red B, el nodo 0 envía y recibe todos los mensajes.

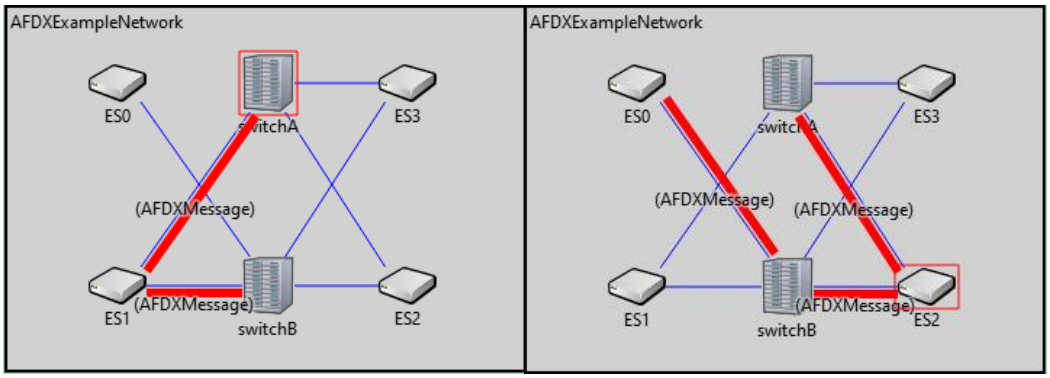


Fig. 2.4 Nodo 1 enviando el mensaje (izquierda) y nodo 0 y nodo 2 recibiendo ese mensaje (derecha)

2.3 CoRE4INET

AFDX y TTEthernet presentan muchas similitudes al estar ambos basados en el estándar IEEE 802.3: ambos utilizan VL en vez de MAC para identificar el destino, la estructura del mensaje es idéntica con los mismas cabeceras y valores máximos y mínimos de carga, etc. La mayor diferencia entre estos dos protocolos es que TTE es capaz de trabajar con 3 tipos de tráfico (TT, RC y BE) mientras que AFDX solo utiliza tráfico RC. Dadas estas similitudes cabe la posibilidad de usar las herramientas propias de la extensión CoRE4INET, utilizada para simular TTE, para crear una red funcional AFDX con la que hacer pruebas. El inconveniente es que no se podrá aplicar la redundancia típica de AFDX ya que la extensión no es compatible, aun siendo TTE compatible con ella.

2.3.1 Red AFDX

Para la realización de las pruebas se ha creado una red AFDX (ver fig. 2.4), con 4 nodos, dos switches y 4 VL para interconectarlos. Todas las VL trabajan con tráfico RC al tratarse de una red AFDX.

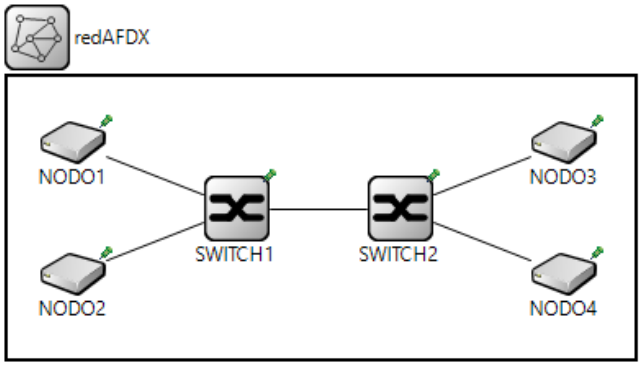


Fig. 2.5 Esquema de la red AFDX

Como en el apartado 1.3.2 ya se explicó el proceso de creación de una con la extensión CoRE4INET, en este apartado sólo se va a concretar sus características. Asimismo, en el apartado 5.3 del anexo está el código de cada fichero utilizado. En la siguiente tabla está la configuración de la comunicación en la red AFDX.

Tabla 2.1 Parámetros de configuración de la red AFDX

VLID	Transmisor	Receptores	Carga del mensaje	BAG	Prioridad
101	Nodo 2	Nodos 1 y 3	46 bytes	1 ms	0
102	Nodo 2	Nodo 1	46 bytes	1 ms	1
103	Nodo 1	Nodos 3 y 4	46 bytes	1 ms	2
104	Nodo 4	Nodos 1 y 3	46 bytes	1 ms	3

Todas las conexiones de la red se han hecho utilizando 100Mbit Ethernet de 10 metros. En las simulaciones para la recogida de datos, se han simulado 100 segundos de funcionamiento de la red.

Una vez la red esté configurada, se puede empezar con las pruebas. Un punto interesante de ver en AFDX es como cada VL se reparte el canal, para configurar este reparto se utiliza la carga del mensaje y el BAG. Normalmente, la carga del mensaje es dependiente de la aplicación que lo genera, por lo que, en el diseño de la red, la variable a definir es únicamente el BAG. Con la configuración definida en la tabla 2.1 obtenemos los siguientes Megabytes transmitidos por el emisor de cada VL durante los 100 segundos simulados.

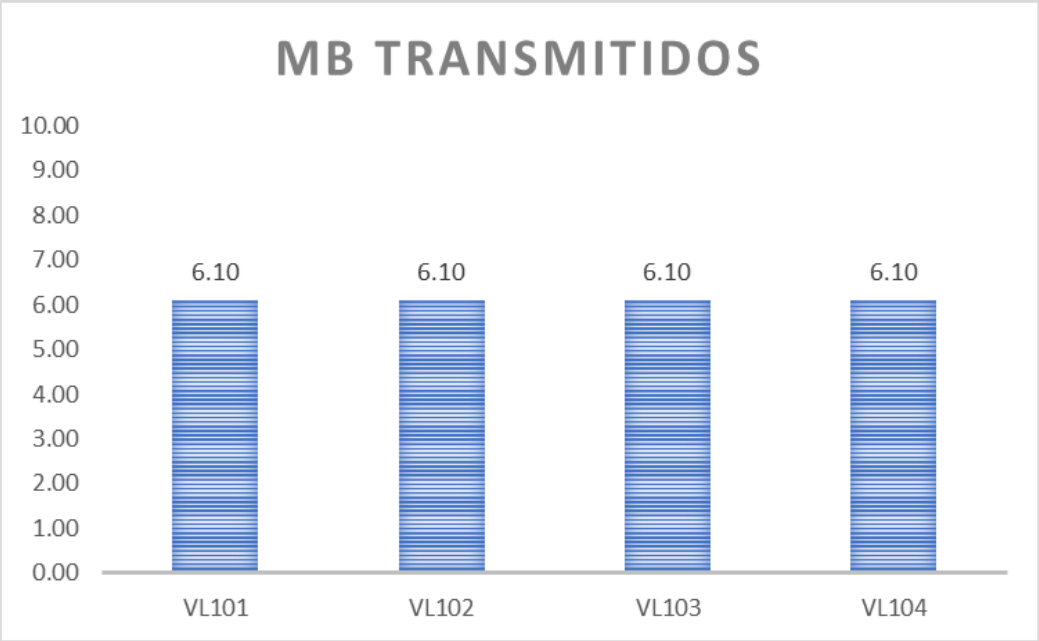


Fig. 2.6 Gráfica de MB transmitidos por el emisor de cada VL

El resultado es el esperable, como todas las VL tienen el mismo BAG y la misma carga, transmiten la misma cantidad de bytes. En una segunda simulación, si cada VL tuviera un valor de carga diferente: VL 101 continua con la carga mínima de 46 bytes, VL 102 tiene una carga de 184 bytes, VL 103 tiene una carga de 736 bytes y VL 104 tiene la carga máxima de 1500 bytes; se puede hacer el ejercicio de intentar repartir equitativamente el canal asignando un BAG proporcional a la carga.

Tabla 2.2 Parámetros para la simulación 2

VLID	Transmisor	Receptores	Carga del mensaje	BAG	Prioridad
101	Nodo 2	Nodos 1 y 3	46 bytes	1 ms	0
102	Nodo 2	Nodo 1	184 bytes	4 ms	1
103	Nodo 1	Nodos 3 y 4	736 bytes	16 ms	2
104	Nodo 4	Nodos 1 y 3	1500 bytes	32 ms	3

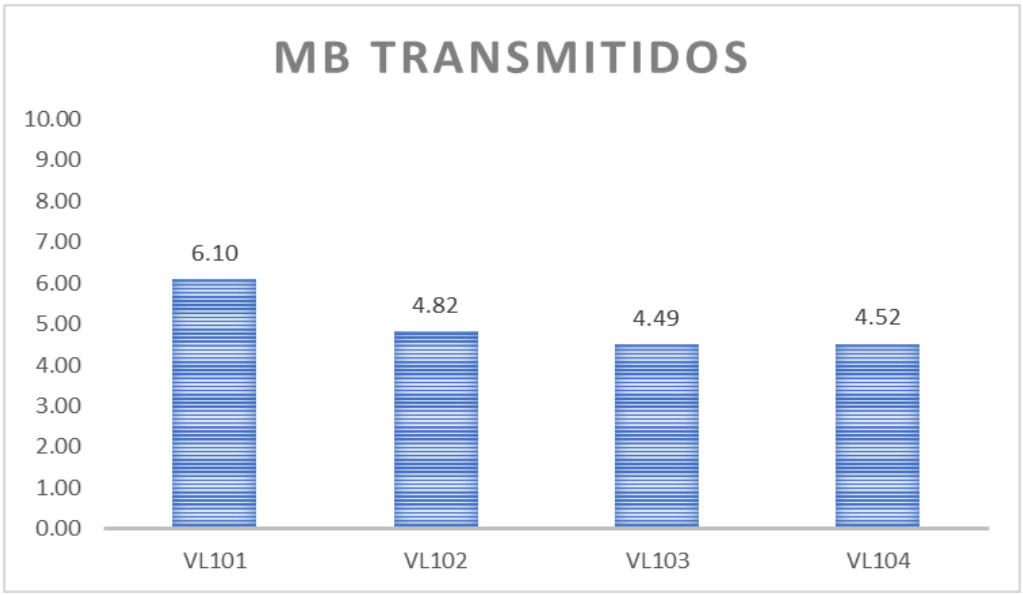


Fig. 2.7 MB transmitidos durante la simulación 2

En la figura 2.7 tenemos los MB transmitidos durante esta segunda simulación. Cabe destacar que un reparto equitativo del canal es imposible si hay diferentes cargas, ya que para el BAG solo se pueden utilizar valores que sean valores que sean potencias de 2 entre 1 a 128 ms.

La definición del BAG no solo afecta a la repartición del canal, sino también a su congestión. Si una red AFDX tiene muchas VL que utilizan la carga máxima del mensaje y tienen un BAG pequeño, puede que el ancho de banda no sea suficiente para afrontar ese tráfico. Este efecto es apreciable midiendo latencias, jitters y paquetes tirados por sobrecarga en buffers de la red. Para recoger datos sobre este efecto se utilizará una tercera simulación con valores máximos de carga y BAG mínimos en cada VL (ver tabla 2.3).

Tabla 2.3 Parámetros para la simulación 3

VLID	Transmisor	Receptores	Carga del mensaje	BAG	Prioridad
101	Nodo 2	Nodos 1 y 3	1500 bytes	1 ms	0
102	Nodo 2	Nodo 1	1500 bytes	1 ms	1
103	Nodo 1	Nodos 3 y 4	1500 bytes	1 ms	2
104	Nodo 4	Nodos 1 y 3	1500 bytes	1 ms	3

Con esta configuración los MB transmitidos por cada VL son 145 MB, todas las VL transmiten lo mismo al tener el mismo BAG y carga. Para calcular la latencia se utilizará todos los valores de latencia recogidos por la simulación desde el emisor al receptor de cada mensaje de cada VL y se hará la media. El jitter se calculará haciendo la deviación estándar de los anteriores valores de latencia. Los mensajes tirados los recoge el simulador.

Tabla 2.4 Resultados de la simulación 3

	Latencia media	Jitter	Mensajes Tirados
Simulación 1	10.4 µs	93.6 ns	0
Simulación 2	85.8 µs	2.28 µs	0
Simulación 3	590 µs	445 µs	0

Estos resultados muestran como para la simulación 3 los valores de latencia y jitter son más altos, debido a la mayor utilización del canal. La simulación 2 ha tenido una latencia y un jitter mayor que la simulación 1 aun teniendo una utilización del canal menor. Esto se puede deber a que en la simulación 1 todos los mensajes tenían la carga mínima, y en la segunda los mensajes con carga más grandes además tienen más prioridad. Estos dos factores juntarse deben augmentar la latencia y el jitter.

Para todas las simulaciones se han tirado cero mensajes por sobrecarga de buffers, la razón se puede obtener con los resultados de la simulación 3. durante la simulación cada VL han transmitido 145 MB, teniendo en cuenta que la simulación ha durado 100 s, la tasa de transmisión es de 1.45 MB/s por VL. Ethernet tiene una capacidad de 100 Mbps, o 12.5 MB/s, por lo que es capaz de soportar el tráfico de 4 VL a máxima tasa de transmisión.

Otra prueba que se le puede hacer a la red es cambiar un VLID para ver como responde la red. Si al nodo 2, el emisor del VL 101, cambiamos el VLID de 101 al 108, una vez el mensaje llegue al switch, este lo tirará ya que no esta configurado para aceptar el VLID 108. Este efecto se puede comprobar con la simulación gráfica, al ver el mensaje desaparecer al llegar al switch, o en los ficheros de resultados, donde un escalar llamado *dropped:count* cuenta cuantos paquetes ha tirado cada dispositivo de la red (ver figura 2.8).

Fig. 2.8 Paquetes tirados por el switch 1 en cada puerto

2.4 Ejemplos y ejercicios académicos

Para aprovechar la extensión AFDX de la que se ha hablado en el punto 2.2, se puede utilizar la red de ejemplo que trae para visualizar el concepto de la redundancia de redes. Para analizar comportamientos de la red se tendrá que volver a trabajar con CoRE4INET, pero a diferencia que con TTE, CoRE4INET no trae ejemplos de redes AFDX, por lo que se deberá crear una de nuevo u

ofrecer una ya construida como la utilizada en el punto 2.3.1, con el código disponible en el anexo 5.3.

Un ejercicio posible con una red AFDX sería, dada una red con un número específico de enlaces VL, cada uno de ellos con una carga de mensaje diferente, configurar la red para repartir de manera equitativa el canal entre todos los VL. Este ejercicio se enfocaría en la asignación de BAGs. Pero hay que tener en cuenta que el reparto equitativo del canal no es posible al tener los valores de BAG limitados, el BAG va de 1 a 128 ms en potencias de 2.

Otro ejercicio posible es dar una lista de aplicaciones, con el nodo transmisor y los nodos receptores, y que el ejercicio conste en diseñar los enlaces VL para comunicar los dispositivos, teniendo en cuenta que hay que hacer el enrutamiento en los switches.

Más usos para la simulación AFDX es crear una red mal configurada intencionadamente, para su posterior análisis para encontrar el error. Por ejemplo, simular la red y ver que el switch tira los paquetes provenientes del nodo 1, y esto se puede deber a distintas razones como que los VLID no están bien configurados, o que la aplicación genera los mensajes con un más carga que la permitida, etc.

Conclusiones

Este proyecto ha conseguido el objetivo principal, poder construir una red con los protocolos AFDX y TTE, y poder modificar sus configuraciones para poder estudiar sus efectos. Todo esto se ha conseguido con simulaciones informáticas, utilizando el software OMNeT++. El software permite crear redes con libertad y configurar todos sus parámetros, además se puede recoger una gran cantidad de datos de cada dispositivo en concreto como, por ejemplo: bits enviados o recibidos, latencia, jitter, mensajes tirados, el tanto por ciento de uso del canal, monitorizado de buffers, etc.

En cuanto a las dificultades afrontadas con el proyecto, en primer lugar, estaba pensado hacer una aplicación física y otra simulada, pero después de ponerme en contacto con una empresa distribuidora de dispositivos aviónicos, TTTech, se tuvo que descartar por temas de presupuesto. Otra dificultad ha sido el aprendizaje de las herramientas del simulador informático, aunque no había utilizado OMNeT++ antes de la realización de este proyecto, al estar muy bien documentado e incluso haber un tutorial[16], ha sido sencillo entender sus bases. El problema ha aparecido al tener que utilizar paquetes de terceros, CoRE4INET y AFDX, los cuales no tienen mucha documentación y ha sido arduo su aprendizaje. Finalmente, también relacionado con el paquete AFDX, este paquete solo simula la capa MAC de la red, y no tiene aplicado el enrutamiento por VL. El autor tiene una anotación donde debería ir el código del enrutamiento que anima al usuario a diseñarla el mismo. Después de muchos intentos por crear una, llegué a la conclusión que mi nivel de C++ no era suficiente para diseñarlo.

El impacto ambiental ha sido casi nulo al haberse basado en simulaciones informáticas, habiendo solo gastado electricidad en procesamiento CPU.

Bibliografía

[1] Muhammet Emin YANIK, “Avionics Full Duplex Switched Ethernet (AFDX) Data Bus”

[2] https://omnetpp.org/download/models-and-tools - Paquetes de terceros para OMNeT++

[3] https://omnetpp.org/download/ - Sitio de descarga de OMNeT++

[4] https://doc.omnetpp.org/omnetpp/InstallGuide.pdf - Guía de instalación de OMNeT++

[5] https://core.informatik.haw-hamburg.de/ - Pagina web del grupo de investigacion CoRE.

[6] https://inet.omnetpp.org/Download.html - Sitio de descarga de INET

[7] https://omnetpp.org/download-items/Afdx.html - Sitio de descarga del paquete AFDX

[8] Paul Grams, NASA, “Ethernet for Aerospace Applications”

[9] W. Steiner, G. Bauer, B. Hall, M. Paulistich & S. Varadarajan, “TTEthernet Data Flow Concept”

[10] Wildfred Steiner & Bruno Dutertre, “Layered Diagnosis and Clock-Rate Correction for the TTEthernet Clock Synchronization Protocol”

[11] T. Steinbach, H. Lim, F. Korf, T. C. Schmidt, D. Herrscher & A. Wolisz, “Tomorrow’s In-Car Interconnect? A competitive Evaluation of IEEE 802.1 AVB and Time-Triggerred Ethernet (AS6802)”

[12] TTTech Computertechnik AG, “TTEthernet – A Powerful Network Solution for All Purposes”

[13] Anna Agustí y José Ramón Piney, “Introducció al Simulador de Xarxes OMNeT++”

[14] Detlev Schaat, “AFDDX/ARINC 664 Concept, Design, Implementation and Beyond”

[15] https://www.aviation-ia.com/content/arinc-standards - Información sobre la estandarización de ARINC

[16] https://docs.omnetpp.org/ - Tutorial de OMNeT++

Anexos

5.1 ARINC y ARINC 429

Aeronautical Radio, Incorporated (ARINC) es la empresa de ingeniería que trabaja los campos de la aviación, aeropuertos, defensa, gobierno, salud, redes y transporte. Una subdelegación de esta empresa, llamada ARINC *Standards*, y con la participación de *Airlines Electronic Engineering Committee* (AEEC), *Aviation Maintenance Conference* (AMC) y *Flight Simulator Engineering and Maintenance Committee* (FSEMC), se encargan de:

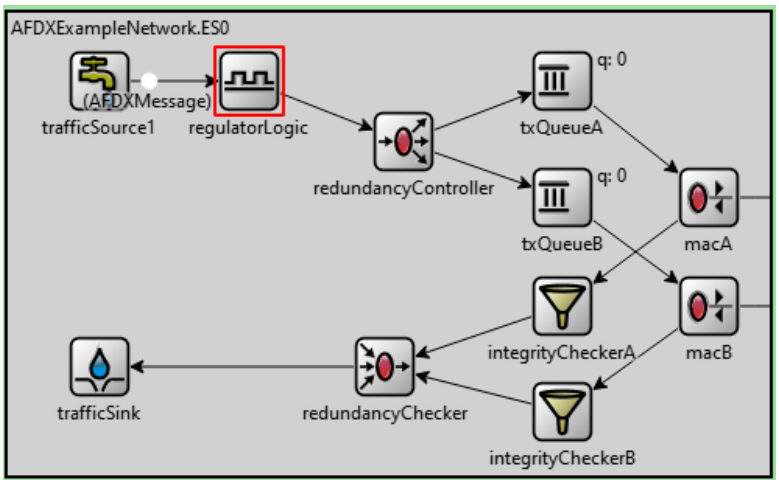
- Definir la forma, el emplazamiento, la función y la interfaz de la aviónica, los sistemas de cabina y las redes aviónicas.
- Define el montaje y el empaquetamiento de la aviónica y el equipo de cabina.
- Define los estándares de comunicación y de seguridad en las redes aviónicas.
- Provee de guías e información sobre la industria de la aviación en temas relacionados con el mantenimiento de los sistemas aviónicos y la confección de simuladores de vuelo.

ARINC 429 es el estándar de bus de comunicación más utilizado en la aviación. Es un bus serie formado por un par trenzado y apantallado, simplex, con capacidad de hasta 20 receptores y un emisor, con una velocidad máxima de transmisión de 100 kbits/s y una topología de despliegue en bus. Su popularidad en el mundo de la aviación se debe a su simpleza, que a su vez conlleva una robustez necesaria en aeronáutica.

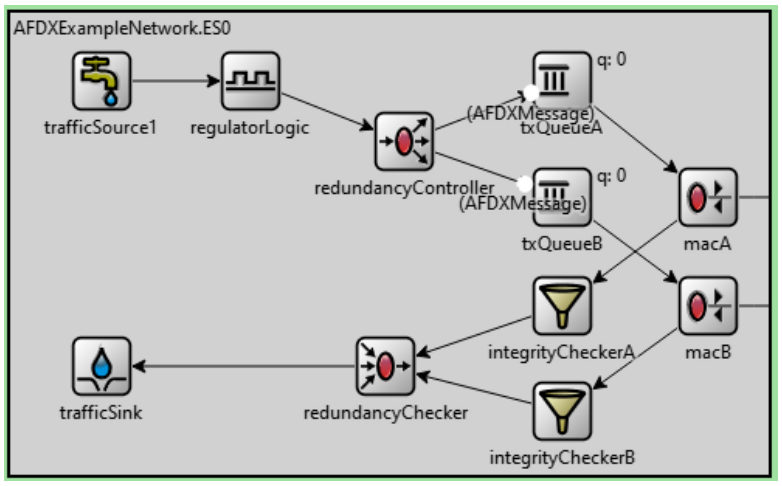
5.2 Tratamiento de la redundancia

En las siguientes ilustraciones se ve como los nodos AFDX tratan la redundancia. Capturas tomadas de la simulación con el paquete AFDX.

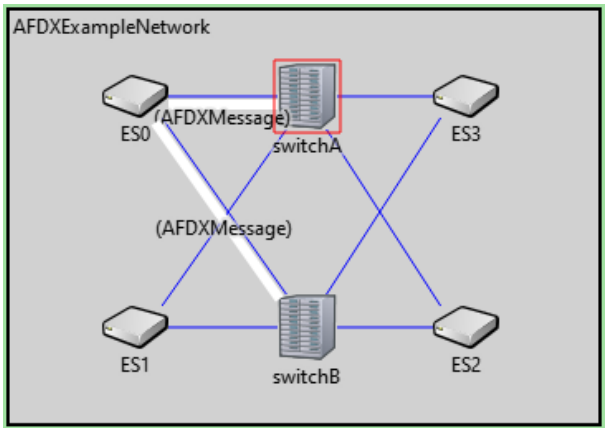
- Se crea un único mensaje:



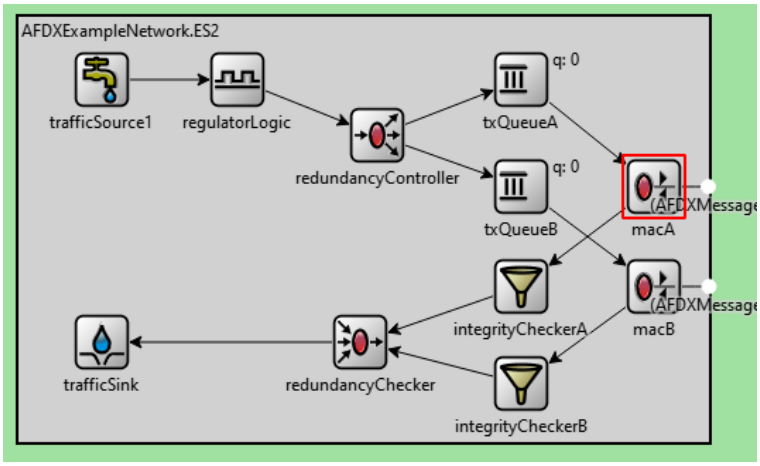
- El controllador de redundancia lo duplica:



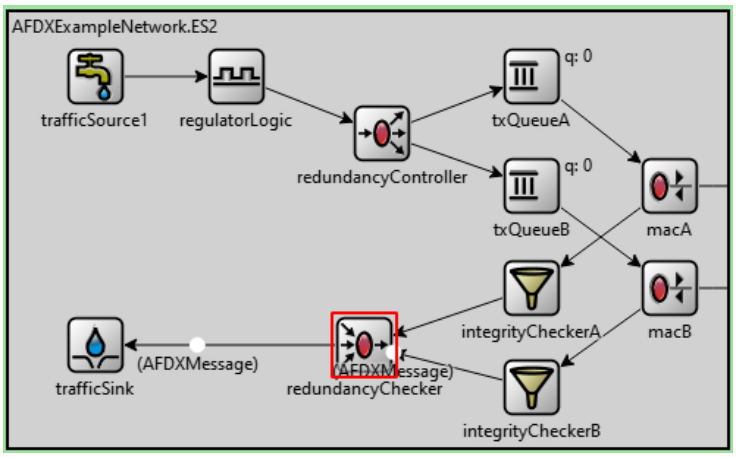
- Cada mensaje es enviado por una red distinta (red A y B):



- Cada nodo recibe dos mensajes identicos:



- El verificador de redundancia elimina uno de ellos y llega al destino:



5.3 Codigo de la red AFDX

En esta parte del anexo está el código de la red utilizada en el apartado 2.3.1 antes de las modificaciones.

Código del archivo NED del nodo 1:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;

module NODO1 extends TTEEtherHost
{
    @display("bgb=503,388");
    submodules:
        // submódulos del VL101
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=350,160");

        }
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=450,160");
        }
        // submódulos del VL102
        vl_102_ctc: RCIncoming {
            parameters:
                @display("p=350,220");

        }
        vl_102: RCQueueBuffer {
            parameters:
                @display("p=450,220");
        }
        // submódulos del VL103
```

```
        vl_103_ctc: RCIncoming {
            parameters:
                @display("p=350,280");

        }
        vl_103: RCQueueBuffer {
            parameters:
                @display("p=450,280");
        }
        // submódulos del VL104
        vl_104_ctc: RCIncoming {
            parameters:
                @display("p=350,340");

        }
        vl_104: RCQueueBuffer {
            parameters:
                @display("p=450,340");
        }

    connections:
        vl_101_ctc.out --> vl_101.in;
        vl_102_ctc.out --> vl_102.in;
        vl_103_ctc.out --> vl_103.in;
        vl_104_ctc.out --> vl_104.in;
}
```

Código del archivo INI del nodo 1:

```
[General]
network = redAFDX

**.NODO1.phy[*].mac.address = "0A-00-00-00-00-01" # MAC del dispositivo

**.NODO1.numApps = 4 # número de aplicaciones

#VL 101 aplicacion de recepcion
**.NODO1.app[0].typename = "CTTrafficSinkApp" # tipo de aplicación
**.NODO1.app[0].displayName = "vl_101" #el nombre
**.NODO1.phy[0].inControl.ct_incomings = "vl_101_ctc" #el controlador
encargado de esta VL
**.NODO1.vl_101_ctc.bag = sec_to_tick(880us)
**.NODO1.vl_101.ct_id = 101 # El VLID que el controlador dejara pasar
**.NODO1.vl_101.destination_gates = "app[0].RCin" # El destino de los
mensajes, la aplicacion para recepcion

#VL 102 aplicacion de recepcion
**.NODO1.app[1].typename = "CTTrafficSinkApp"
**.NODO1.app[1].displayName = "vl_102"
**.NODO1.phy[0].inControl.ct_incomings = "vl_102_ctc"
**.NODO1.vl_102_ctc.bag = sec_to_tick(880us)
**.NODO1.vl_102.ct_id = 102
**.NODO1.vl_102.destination_gates = "app[1].RCin"

#VL 103 aplicacion de transmisión
**.NODO1.app[2].typename = "RCTrafficSourceApp" #tipo de aplicación
**.NODO1.app[2].displayName = "vl_103" # el nombre
**.NODO1.app[2].interval = 1ms # el valor del BAG
**.NODO1.app[2].payload = 46Byte # la carga del mensaje de 46 a 1500 bytes
**.NODO1.app[2].ct_id = 103 # el VLID
**.NODO1.app[2].buffers = "vl_103" # El bufffer encargado de esta VL
**.NODO1.vl_103.destination_gates = "phy[0].RCin" # el destino, la tarjeta
física para transmisión
**.NODO1.vl_103.bag = sec_to_tick(900us)
**.NODO1.vl_103.priority = 2 # la prioridad del mensaje (más alto, más
prioridad)
**.NODO1.vl_103.ct_id = 103 # el VLID que el controlador dejará pasar

**.NODO1.app[3].typename = "CTTrafficSinkApp"
**.NODO1.app[3].displayName = "vl_104"
**.NODO1.phy[0].inControl.ct_incomings = "vl_104_ctc"
**.NODO1.vl_104_ctc.bag = sec_to_tick(880us)
**.NODO1.vl_104.ct_id = 104
**.NODO1.vl_104.destination_gates = "app[3].RCin"
```

Código del archivo NED del nodo 2:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;

module NODO2 extends TTEEtherHost
{
    @display("bgb=503,320");
    submodules:
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=350,160");

        }
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=450,160");
        }
        vl_102_ctc: RCIncoming {
            parameters:
                @display("p=350,220");

        }
        vl_102: RCQueueBuffer {
            parameters:
                @display("p=450,220");
        }

    connections:
        vl_101_ctc.out --> vl_101.in;
        vl_102_ctc.out --> vl_102.in;
}
```

Código del archivo INI del nodo 2:

```
[General]
network = redAFDX

**.NODO2.phy[*].mac.address = "0A-00-00-00-00-02"

**.NODO2.numApps = 2

**.NODO2.app[0].typename = "RCTrafficSourceApp"
**.NODO2.app[0].displayName = "vl_101"
**.NODO2.app[0].interval = 1ms
**.NODO2.app[0].payload = 46Byte
**.NODO2.app[0].ct_id = 101
**.NODO2.app[0].buffers = "vl_101"
**.NODO2.vl_101.destination_gates = "phy[0].RCin"
**.NODO2.vl_101.bag = sec_to_tick(900us)
**.NODO2.vl_101.priority = 0
**.NODO2.vl_101.ct_id = 101

**.NODO2.app[1].typename = "RCTrafficSourceApp"
**.NODO2.app[1].displayName = "vl_102"
**.NODO2.app[1].interval = 1ms
**.NODO2.app[1].payload = 46Byte
**.NODO2.app[1].ct_id = 102
**.NODO2.app[1].buffers = "vl_102"
**.NODO2.vl_102.destination_gates = "phy[0].RCin"
**.NODO2.vl_102.bag = sec_to_tick(900us)
**.NODO2.vl_102.priority = 1
**.NODO2.vl_102.ct_id = 102
```

Código del archivo NED del nodo 3:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;

module NODO3 extends TTEEtherHost
{
    @display("bgb=503,314");
    submodules:
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=350,160");

        }
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=450,160");
        }
        vl_103_ctc: RCIncoming {
            parameters:
                @display("p=350,220");

        }
        vl_103: RCQueueBuffer {
            parameters:
                @display("p=450,220");
        }
        vl_104_ctc: RCIncoming {
            parameters:
                @display("p=350,280");

        }
        vl_104: RCQueueBuffer {
            parameters:
                @display("p=450,280");
        }
    connections:
        vl_101_ctc.out --> vl_101.in;
        vl_103_ctc.out --> vl_103.in;
        vl_104_ctc.out --> vl_104.in;
}
```

Código del archivo INI del nodo 3:

```
[General]
network = redAFDX

**.NODO3.phy[*].mac.address = "0A-00-00-00-00-03"

**.NODO3.numApps = 3

**.NODO3.app[0].typename = "CTTrafficSinkApp"
**.NODO3.app[0].displayName = "vl_101"
**.NODO3.phy[0].inControl.ct_incomings = "vl_101_ctc"
**.NODO3.vl_101_ctc.bag = sec_to_tick(880us)
**.NODO3.vl_101.ct_id = 101
**.NODO3.vl_101.destination_gates = "app[0].RCin"

**.NODO3.app[1].typename = "CTTrafficSinkApp"
**.NODO3.app[1].displayName = "vl_103"
**.NODO3.phy[1].inControl.ct_incomings = "vl_103_ctc"
**.NODO3.vl_103_ctc.bag = sec_to_tick(880us)
**.NODO3.vl_103.ct_id = 103
**.NODO3.vl_103.destination_gates = "app[1].RCin"

**.NODO3.app[2].typename = "CTTrafficSinkApp"
**.NODO3.app[2].displayName = "vl_104"
**.NODO3.phy[2].inControl.ct_incomings = "vl_104_ctc"
**.NODO3.vl_104_ctc.bag = sec_to_tick(880us)
**.NODO3.vl_104.ct_id = 104
**.NODO3.vl_104.destination_gates = "app[2].RCin"
```

Código del archivo NED del nodo 4:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherHost;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;

module NODO4 extends TTEEtherHost
{
    @display("bgb=503,314");
    submodules:
        vl_103_ctc: RCIncoming {
            parameters:
                @display("p=350,220");

        }
        vl_103: RCQueueBuffer {
            parameters:
                @display("p=450,220");
        }
        vl_104_ctc: RCIncoming {
            parameters:
                @display("p=350,160");

        }
        vl_104: RCQueueBuffer {
            parameters:
                @display("p=450,160");
        }
    connections:
        vl_103_ctc.out --> vl_103.in;
        vl_104_ctc.out --> vl_104.in;
}
```

Código del archivo INI del nodo 4:

```
[General]
network = redAFDX

**.NODO4.phy[*].mac.address = "0A-00-00-00-00-04"

**.NODO4.numApps = 2

**.NODO4.app[0].typename = "CTTrafficSinkApp"
**.NODO4.app[0].displayName = "vl_103"
**.NODO4.phy[0].inControl.ct_incomings = "vl_103_ctc"
**.NODO4.vl_103_ctc.bag = sec_to_tick(880us)
**.NODO4.vl_103.ct_id = 103
**.NODO4.vl_103.destination_gates = "app[0].RCin"

**.NODO4.app[1].typename = "RCTrafficSourceApp"
**.NODO4.app[1].displayName = "vl_104"
**.NODO4.app[1].interval = 1ms
**.NODO4.app[1].payload = 46Byte
**.NODO4.app[1].ct_id = 104
**.NODO4.app[1].buffers = "vl_104"
**.NODO4.vl_104.destination_gates = "phy[0].RCin"
**.NODO4.vl_104.bag = sec_to_tick(900us)
**.NODO4.vl_104.priority = 3
**.NODO4.vl_104.ct_id = 104
```

Código del archivo NED del switch 1:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherSwitch;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;


module SWITCH1 extends TTEEtherSwitch
{
    @display("bgb=500,381");
    submodules:
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=100,200");
                hardware_delay = hardware_delay;
        }
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=200,200");
        }
        vl_102_ctc: RCIncoming {
            parameters:
                @display("p=100,260");
                hardware_delay = hardware_delay;
        }
        vl_102: RCQueueBuffer {
            parameters:
                @display("p=200,260");
        }
        vl_103_ctc: RCIncoming {
            parameters:
                @display("p=300,200");

        }
        vl_103: RCQueueBuffer {
            parameters:
                @display("p=400,200");
        }
        vl_104_ctc: RCIncoming {
            parameters:
                @display("p=300,260");

        }
        vl_104: RCQueueBuffer {
            parameters:
                @display("p=400,260");
        }
    connections:
        vl_101_ctc.out --> vl_101.in;
        vl_102_ctc.out --> vl_102.in;
        vl_103_ctc.out --> vl_103.in;
        vl_104_ctc.out --> vl_104.in;
}
```

Código del archivo INI del switch 1:

```
[General]
network = redAFDX

# indicar en cada puerto del switch que controladores trabajan
**.SWITCH1.phy[0].inControl.ct_incomings ="vl_101_ctc, vl_102_ctc,
vl_103_ctc, vl_104_ctc"
**.SWITCH1.phy[1].inControl.ct_incomings = "vl_101_ctc, vl_102_ctc,
vl_103_ctc, vl_104_ctc"
**.SWITCH1.phy[2].inControl.ct_incomings = "vl_101_ctc, vl_102_ctc,
vl_103_ctc, vl_104_ctc"

**.SWITCH1.vl_101_ctc.bag = sec_to_tick(880us)
**.SWITCH1.vl_101.destination_gates = "phy[0].RCin, phy[2].RCin"# a que
puertos tiene que ir el VL101
**.SWITCH1.vl_101.bag = sec_to_tick(900us)
**.SWITCH1.vl_101.priority = 0 # la prioridad del mensaje
**.SWITCH1.vl_101.ct_id = 101 # el VLID que el controlador dejará pasar

**.SWITCH1.vl_102_ctc.bag = sec_to_tick(880us)
**.SWITCH1.vl_102.destination_gates = "phy[0].RCin"# a que puertos tiene que
ir el VL102
**.SWITCH1.vl_102.bag = sec_to_tick(900us)
**.SWITCH1.vl_102.priority = 1# la prioridad del mensaje
**.SWITCH1.vl_102.ct_id = 102# el VLID que el controlador dejará pasar

**.SWITCH1.vl_103_ctc.bag = sec_to_tick(880us)
**.SWITCH1.vl_103.destination_gates = "phy[2].RCin"# a que puertos tiene que
ir el VL103
**.SWITCH1.vl_103.bag = sec_to_tick(900us)
**.SWITCH1.vl_103.priority = 2# la prioridad del mensaje
**.SWITCH1.vl_103.ct_id = 103# el VLID que el controlador dejará pasar

**.SWITCH1.vl_104_ctc.bag = sec_to_tick(880us)
**.SWITCH1.vl_104.destination_gates = "phy[0].RCin"# a que puertos tiene que
ir el VL104
**.SWITCH1.vl_104.bag = sec_to_tick(900us)
**.SWITCH1.vl_104.priority = 3# la prioridad del mensaje
**.SWITCH1.vl_104.ct_id = 104# el VLID que el controlador dejará pasar
```

Código del archivo NED del switch 2:

```
import core4inet.nodes.ethernet.AS6802.TTEEtherSwitch;
import core4inet.incoming.AS6802.TTIncoming;
import core4inet.buffer.AS6802.TTDoubleBuffer;
import core4inet.incoming.AS6802.RCIncoming;
import core4inet.buffer.AS6802.RCQueueBuffer;


module SWITCH2 extends TTEEtherSwitch
{
    @display("bgb=500,351");
    submodules:
        vl_101_ctc: RCIncoming {
            parameters:
                @display("p=100,200");
                hardware_delay = hardware_delay;
        }
        vl_101: RCQueueBuffer {
            parameters:
                @display("p=200,200");
        }
        vl_103_ctc: RCIncoming {
            parameters:
                @display("p=300,200");

        }
        vl_103: RCQueueBuffer {
            parameters:
                @display("p=400,200");
        }
        vl_104_ctc: RCIncoming {
            parameters:
                @display("p=300,260");

        }
        vl_104: RCQueueBuffer {
            parameters:
                @display("p=400,260");
        }
    connections:
        vl_101_ctc.out --> vl_101.in;
        vl_103_ctc.out --> vl_103.in;
        vl_104_ctc.out --> vl_104.in;
}
```

Código del archivo INI del switch 2:

```
[General]
network = redAFDX

**.SWITCH2.phy[0].inControl.ct_incomings = "vl_101_ctc, vl_103_ctc,
vl_104_ctc"
**.SWITCH2.phy[1].inControl.ct_incomings = "vl_101_ctc, vl_103_ctc,
vl_104_ctc"
**.SWITCH2.phy[2].inControl.ct_incomings = "vl_101_ctc, vl_103_ctc,
vl_104_ctc"

**.SWITCH2.vl_101_ctc.bag = sec_to_tick(880us)
**.SWITCH2.vl_101.destination_gates = "phy[1].RCin "
**.SWITCH2.vl_101.bag = sec_to_tick(900us)
**.SWITCH2.vl_101.priority = 0
**.SWITCH2.vl_101.ct_id = 101

**.SWITCH2.vl_103_ctc.bag = sec_to_tick(880us)
**.SWITCH2.vl_103.destination_gates = "phy[0].RCin,phy[1].RCin "
**.SWITCH2.vl_103.bag = sec_to_tick(900us)
**.SWITCH2.vl_103.priority = 2
**.SWITCH2.vl_103.ct_id = 103

**.SWITCH2.vl_104_ctc.bag = sec_to_tick(880us)
**.SWITCH2.vl_104.destination_gates = "phy[0].RCin,phy[2].RCin "
**.SWITCH2.vl_104.bag = sec_to_tick(900us)
**.SWITCH2.vl_104.priority = 3
**.SWITCH2.vl_104.ct_id = 104
```

Código del archivo NED de la red AFDX:

```
import inet.node.ethernet.Eth100M;


network redAFDX
{
    @display("bgb=,,white");
    submodules:
        NODO1: NODO1 {
            @display("p=40,40");
        }
        NODO2: NODO2 {
            @display("p=40,120");
        }
        NODO3: NODO3 {
            @display("p=340,40");
        }
        NODO4: NODO4 {
            @display("p=340,120");
        }
        SWITCH1: SWITCH1 {
            parameters:
                @display("p=140,80");
            gates:
                ethg[3];
        }
        SWITCH2: SWITCH2 {
            parameters:
                @display("p=240,80");
            gates:
                ethg[3];
        }
    connections:
        NODO1.ethg <--> Eth100M {  length = 10m; } <--> SWITCH1.ethg[0];
        NODO2.ethg <--> Eth100M {  length = 10m; } <--> SWITCH1.ethg[1];
        SWITCH2.ethg[2] <--> Eth100M {  length = 10m; } <--> SWITCH1.ethg[2];
        NODO3.ethg <--> Eth100M {  length = 10m; } <--> SWITCH2.ethg[0];
        NODO4.ethg <--> Eth100M {  length = 10m; } <--> SWITCH2.ethg[1];
}
```

Código del archivo omnet.ini:

```
[General]
check-signals = true
tkenv-plugin-path = ../../../etc/plugins
**.vector-recording = true

**.scalar-recording = true

network = redAFDX

# Global config
**.ct_marker = 0x03040506
**.ct_mask = 0xffffffff


include NODO1.ini
include NODO2.ini
include NODO3.ini
include NODO4.ini
include SWITCH1.ini
include SWITCH2.ini
```